Philip Wadler, Wen Kokke, and Jeremy G. Siek

PROGRAMMING LANGUAGE FOUNDATIONS IN Agda

2021

Contents

Dedication

de Philip, para Wanda

amor da minha vida

knock knock knock

...

Preface

The most profound connection between logic and computation is a pun. The doctrine of Propositions as Types asserts that a certain kind of formal structure may be read in two ways: either as a proposition in logic or as a type in computing. Further, a related structure may be read as either the proof of the proposition or as a programme of the corresponding type. Further still, simplification of proofs corresponds to evaluation of programs.

Accordingly, the title of this book also has two readings. It may be parsed as "(Programming Language) Foundations in Agda" or "Programming (Language Foundations) in Agda" — the specifications we will write in the proof assistant Agda both describe programming languages and are themselves programmes.

The book is aimed at students in the last year of an undergraduate honours programme or the first year of a master or doctorate degree. It aims to teach the fundamentals of operational semantics of programming languages, with simply-typed lambda calculus as the central example. The textbook is written as a literate script in Agda. The hope is that using a proof assistant will make the development more concrete and accessible to students, and give them rapid feedback to find and correct misapprehensions.

The book is broken into two parts. The first part, Logical Foundations, develops the needed formalisms. The second part, Programming Language Foundations, introduces basic methods of operational semantics.

Personal remarks

Since 2013, I have taught a course on Types and Semantics for Programming Languages to fourth-year undergraduates and masters students at the University of Edinburgh. An earlier version of that course was based on Benjamin Pierce's excellent TAPL. My version was based of Pierce's subsequent textbook, Software Foundations, written in collaboration with others and based on Coq. I am convinced of Pierce's claim that basing a course around a proof assistant aids learning, as summarised in his ICFP Keynote, Lambda, The Ultimate TA.

However, after five years of experience, I have come to the conclusion that Coq is not the best vehicle. Too much of the course needs to focus on learning tactics for proof derivation, to the cost of learning the fundamentals of programming language theory. Every concept has to be learned twice: e.g., both the product data type, and the corresponding tactics for introduction and elimination of conjunctions. The rules Coq applies to generate induction hypotheses can sometimes seem mysterious. While the `notation` construct permits pleasingly flexible syntax, it can be confusing that the same concept must always be given two names, e.g., both `subst N x M` and `N [x := M]`. Names of tactics are sometimes short and sometimes long; naming conventions in the standard library can be wildly inconsistent. *Propositions as types* as a foundation of proof is present but hidden.

I found myself keen to recast the course in Agda. In Agda, there is no longer any need to learn about tactics: there is just dependently-typed programming, plain and simple. Introduction is always by a constructor, elimination is always by pattern matching. Induction is no longer a

mysterious separate concept, but corresponds to the familiar notion of recursion. Mixfix syntax is flexible while using just one name for each concept, e.g., substitution is `_[_:=_]`. The standard library is not perfect, but there is a fair attempt at consistency. *Propositions as types* as a foundation of proof is on proud display.

Alas, there is no textbook for programming language theory in Agda. Stump's Verified Functional Programming in Agda covers related ground, but focusses more on programming with dependent types than on the theory of programming languages.

The original goal was to simply adapt *Software Foundations*, maintaining the same text but transposing the code from Coq to Agda. But it quickly became clear to me that after five years in the classroom I had my own ideas about how to present the material. They say you should never write a book unless you cannot *not* write the book, and I soon found that this was a book I could not not write.

I am fortunate that my student, Wen Kokke, was keen to help. She guided me as a newbie to Agda and provided an infrastructure that is easy to use and produces pages that are a pleasure to view.

Most of the text was written during a sabbatical in the first half of 2018.

— Philip Wadler, Rio de Janeiro, January–June 2018

A word on the exercises

Exercises labelled "(recommended)" are the ones students are required to do in the class taught at Edinburgh from this textbook.

Exercises labelled "(stretch)" are there to provide an extra challenge. Few students do all of these, but most attempt at least a few.

Exercises labelled "(practice)" are included for those who want extra practice.

You may need to import library functions required for the solution.

Please do not post answers to the exercises in a public place.

There is a private repository of answers to selected questions on github. Please contact Philip Wadler if you would like to access it.

Getting Started

Dependencies for users

You can read PLFA online without installing anything. However, if you wish to interact with the code or complete the exercises, you need several things:

- Stack
- Git
- Agda
- Agda standard library
- PLFA

PLFA is tested against specific versions of Agda and the standard library, which are shown in the badges above. Agda and the standard library change rapidly, and these changes often break PLFA, so using older or newer versions usually causes problems.

There are several versions of Agda and its standard library online. If you are using a package manager, like Homebrew or Debian apt, the version of Agda available there may be out-of date. Furthermore, Agda is under active development, so if you install the development version from the GitHub, you might find the developers have introduced changes which break the code here. Therefore, it's important to have the specific versions of Agda and the standard library shown above.

On macOS: Install the XCode Command Line Tools

On macOS, you'll need to install the XCode Command Line Tools. For most versions of macOS, you can install these by running the following command:

```
xcode-select --install
```

Install the Haskell Tool Stack

Agda is written in Haskell, so to install it we'll need the *Haskell Tool Stack*, or *Stack* for short. Stack is a program for managing different Haskell compilers and packages:

- *On UNIX and macOS.* If your package manager has a package for Stack, that's probably your easiest option. For instance, Homebrew on macOS and APT on Debian offer the "haskell-stack" package. Otherwise, you can follow the instructions on the Stack website. Usually, Stack installs binaries at `HOME/.local/bin`. Please ensure this is on your PATH, by including the following in your shell configuration, e.g., in `HOME/.bash_profile`:

```
export PATH="${HOME}/.local/bin:${PATH}"
```

Finally, ensure that you've got the latest version of Stack, by running:

```
stack upgrade
```

- *On Windows.* There is a Windows installer on the Stack website.

Install Git

If you do not already have Git installed, see the Git downloads page.

Install Agda using Stack

The easiest way to install a *specific version* of Agda is using Stack. You can get the required version of Agda from GitHub, either by cloning the repository and switching to the correct branch, or by downloading the zip archive:

```
git clone https://github.com/agda/agda.git
cd agda
git checkout v2.6.1.3
```

To install Agda, run Stack from the Agda source directory:

```
stack install --stack-yaml stack-8.8.3.yaml
```

This step will take a long time and a lot of memory to complete.

Using an existing installation of GHC

Stack is perfectly capable of installing and managing versions of the Glasgow Haskell Compiler for you. However, if you already have a copy of GHC installed, and you want Stack to use your system installation, you can pass the `--system-ghc` flag and select the appropriate `stack-*.yaml` file. For instance, if you have GHC 8.2.2 installed, run:

```
stack install --system-ghc --stack-yaml stack-8.2.2.yaml
```

Check if Agda was installed correctly

If you'd like, you can test to see if you've installed Agda correctly. Create a file called `hello.agda` with these lines:

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

From a command line, change to the same directory where your `hello.agda` file is located. Then run:

```
agda -v 2 hello.agda
```

You should see a short message like the following, but no errors:

```
Checking hello (/path/to/hello.agda).
Finished hello.
```

Install PLFA and the Agda standard library

You can get the latest version of Programming Language Foundations in Agda from GitHub, either by cloning the repository, or by downloading the zip archive:

```
git clone --depth 1 --recurse-submodules --shallow-submodules https://github.com/plfa/plfa.gith
# Remove `--depth 1` and `--shallow-submodules` if you want the complete git history of PLFA an
```

PLFA ships with the required version of the Agda standard library, so if you cloned with the `--recurse-submodules` flag, you've already got, in the `standard-library` directory!

If you forgot to add the `--recurse-submodules` flag, no worries, we can fix that!

```
cd plfa/
git submodule update --init --recursive --depth 1
# Remove `--depth 1` if you want the complete git history of the standard library.
```

If you obtained PLFA by downloading the zip archive, you can get the required version of the Agda standard library from GitHub. You can either clone the repository and switch to the correct branch, or you can download the the zip archive:

```
git clone https://github.com/agda/agda-stdlib.git --branch v1.6 --depth 1 agda-stdlib
# Remove `--depth 1` if you want the complete git history of the standard library.
```

Finally, we need to let Agda know where to find the Agda standard library. You'll need the path where you installed the standard library. Check to see that the file "standard-library.agda-lib" exists, and make a note of the path to this file. You will need to create two configuration files in `AGDA_DIR`. On UNIX and macOS, `AGDA_DIR` defaults to `~/.agda`. On Windows, `AGDA_DIR` usually defaults to `%AppData%\agda`, where `%AppData%` usually defaults to `C:\Users\USERNAME\AppData\Roaming`.

- If the `AGDA_DIR` directory does not already exist, create it.
- In `AGDA_DIR`, create a plain-text file called `libraries` containing the `/path/to/standard-library.agda-lib`. This lets Agda know that an Agda library called `standard-library` is available.
- In `AGDA_DIR`, create a plain-text file called `defaults` containing *just* the line `standard-library`.

More information about placing the standard libraries is available from the Library Management page of the Agda documentation.

It is possible to set up PLFA as an Agda library as well. If you want to complete the exercises found in the `courses` folder, or to import modules from the book, you need to do this. To do so, add the path to `plfa.agda-lib` to `AGDA_DIR/libraries` and add `plfa` to `AGDA_DIR/defaults`, each on a line of their own.

Check if the Agda standard library was installed correctly

If you'd like, you can test to see if you've installed the Agda standard library correctly. Create a file called `nats.agda` with these lines:

```
open import Data.Nat

ten : ℕ
ten = 10
```

(Note that the ℕ is a Unicode character, not a plain capital N. You should be able to just copy-and-paste it from this page into your file.)

From a command line, change to the same directory where your `nats.agda` file is located. Then run:

```
agda -v 2 nats.agda
```

You should see a several lines describing the files which Agda loads while checking your file, but no errors:

```
Checking nats (/path/to/nats.agda).
Loading  Agda.Builtin.Equality (…).
…
Loading  Data.Nat (…).
Finished nats.
```

Setting up an editor for Agda

Emacs

The recommended editor for Agda is Emacs. To install Emacs:

- *On UNIX*, the version of Emacs in your repository is probably fine as long as it is fairly recent. There are also links to the most recent release on the GNU Emacs downloads page.
- *On MacOS*, Aquamacs is probably the preferred version of Emacs, but GNU Emacs can also be installed via Homebrew or MacPorts. See the GNU Emacs downloads page for instructions.
- *On Windows*. See the GNU Emacs downloads page for instructions.

Make sure that you are able to open, edit, and save text files with your installation. The tour of Emacs page on the GNU Emacs site describes how to access the tutorial within your Emacs installation.

Agda ships with the editor support for Emacs built-in, so if you've installed Agda, all you have to do to configure Emacs is run:

```
agda-mode setup
agda-mode compile
```

If you are already an Emacs user and have customized your setup, you may want to note the configuration which the `setup` appends to your `.emacs` file, and integrate it with your own preferred setup.

Check if `agda-mode` was installed correctly

Open the `nats.agda` file you created earlier, and load and type-check the file by typing `C-c C-l`.

Auto-loading `agda-mode` in Emacs

Since version 2.6.0, Agda has had support for literate editing with Markdown, using the `.lagda.md` extension. One issue is that Emacs will default to Markdown editing mode for files with a `.md` suffix. In order to have `agda-mode` automatically loaded whenever you open a file ending with `.agda` or `.lagda.md`, add the following line to your Emacs configuration file:

```
;; auto-load agda-mode for .agda and .lagda.md
(setq auto-mode-alist
   (append
     '(("\\.agda\\'" . agda2-mode)
       ("\\.lagda.md\\'" . agda2-mode))
     auto-mode-alist))
```

If you already have settings which change your `auto-mode-alist` in your configuration, put these *after* the ones you already have or combine them if you are comfortable with Emacs Lisp. The configuration file for Emacs is normally located in `HOME/.emacs` or `HOME/.emacs.d/init.el`, but Aquamacs users might need to move their startup settings to the "Preferences.el" file in `HOME/Library/Preferences/Aquamacs Emacs/Preferences`. For Windows, see the GNU Emacs documentation for a description of where the Emacs configuration is located.

Optional: using the mononoki font with Emacs

Agda uses Unicode characters for many key symbols, and it is important that the font which you use to view and edit Agda programs shows these symbols correctly. The most important part is that the font you use has good Unicode support, so while we recommend mononoki, fonts such as Source Code Pro, DejaVu Sans Mono, and FreeMono are all good alternatives.

You can download and install mononoki directly from GitHub. For most systems, installing a font is merely a matter of clicking the downloaded `.otf` or `.ttf` file. If your package manager offers a package for mononoki, that might be easier. For instance, Homebrew on macOS offers the `font-mononoki` package in the `cask-fonts` cask, and APT on Debian offers the `fonts-mononoki` package. To configure Emacs to use mononoki as its default font, add the following to the end of your Emacs configuration file:

```
;; default to mononoki
(set-face-attribute 'default nil
                    :family "mononoki"
                    :height 120
                    :weight 'normal
                    :width  'normal)
```

Using `agda-mode` in Emacs

To load and type-check the file, use `C-c C-l`.

Agda is edited interactively, using "holes", which are bits of the program that are not yet filled in. If you use a question mark as an expression, and load the buffer using `C-c C-l`, Agda replaces the question mark with a hole. There are several things you can to while the cursor is in a hole:

- `C-c C-c`: **c**ase split (asks for variable name)
- `C-c C-space`: fill in hole
- `C-c C-r`: **r**efine with constructor
- `C-c C-a`: **a**utomatically fill in hole
- `C-c C-,`: goal type and context
- `C-c C-.`: goal type, context, and inferred type

See the emacs-mode docs for more details.

If you want to see messages beside rather than below your Agda code, you can do the following:

- Open your Agda file, and load it using `C-c C-l`;
- type `C-x 1` to get only your Agda file showing;
- type `C-x 3` to split the window horizontally;
- move your cursor to the right-hand half of your frame;
- type `C-x b` and switch to the buffer called "Agda information".

Now, error messages from Agda will appear next to your file, rather than squished beneath it.

Entering Unicode characters in Emacs with `agda-mode`

When you write Agda code, you will need to insert characters which are not found on standard keyboards. Emacs "agda-mode" makes it easier to do this by defining character translations: when you enter certain sequences of ordinary characters (the kind you find on any keyboard), Emacs will replace them in your Agda file with the corresponding special character.

For example, we can add a comment line to one of the `.agda` test files. Let's say we want to add a comment line that reads:

```
{- I am excited to type ∀ and → and ≤ and ≡ !! -}
```

The first few characters are ordinary, so we would just type them as usual...

```
{- I am excited to type
```

But after that last space, we do not find ∀ on the keyboard. The code for this character is the four characters `\all`, so we type those four characters, and when we finish, Emacs will replace them with what we want...

```
{- I am excited to type ∀
```

We can continue with the codes for the other characters. Sometimes the characters will change as we type them, because a prefix of our character's code is the code of another character. This happens with the arrow, whose code is `\->`. After typing `\-` we see...

```
{- I am excited to type ∀ and
```

...because the code `\-` corresponds to a hyphen of a certain width. When we add the `>`, the ‑ becomes `→`! The code for `≤` is `\<=`, and the code for `≡` is `\==`.

```
{- I am excited to type ∀ and → and ≤ and ≡
```

Finally the last few characters are ordinary again...

```
{- I am excited to type ∀ and → and ≤ and ≡ !! -}
```

If you're having trouble typing the Unicode characters into Emacs, the end of each chapter should provide a list of the Unicode characters introduced in that chapter.

Emacs with `agda-mode` offers a number of useful commands, and two of them are especially useful when it comes to working with Unicode characters. For a full list of supported characters, use `agda-input-show-translations` with:

```
M-x agda-input-show-translations
```

All the supported characters in `agda-mode` are shown.

If you want to know how you input a specific Unicode character in agda file, move the cursor onto the character and type the following command:

```
M-x quail-show-key
```

You'll see the key sequence of the character in mini buffer.

Spacemacs

Spacemacs is a "community-driven Emacs distribution" with native support for both Emacs and Vim editing styles. It comes with integration for `agda-mode` out of the box. All that is required is that you turn it on.

Visual Studio Code

Visual Studio Code is a free source code editor developed by Microsoft. There is a plugin for Agda support available on the Visual Studio Marketplace.

Atom

Atom is a free source code editor developed by GitHub. There is a plugin for Agda support available on the Atom package manager.

Dependencies for developers

PLFA is written in literate Agda with Pandoc Markdown. PLFA is available as both a website and an EPUB e-book, both of which can be built on UNIX and macOS. Finally, to help developers avoid common mistakes, we provide a set of Git hooks.

Building the website and e-book

If you'd like to build the web version of PLFA locally, Stack is all you need! PLFA is built using Hakyll, a Haskell library for building static websites. We've setup a Makefile to help you run common tasks. For instance, to build PLFA, run:

```
make build
```

If you'd like to serve PLFA locally, rebuilding the website when any of the source files are changed, run:

```
make watch
```

The Makefile offers more than just building and watching, it also offers the following useful options:

```
build                     # Build PLFA
watch                     # Build and serve PLFA, monitor for changes and rebuild
test                      # Test web version for broken links, invalid HTML, etc.
test-epub                 # Test EPUB for compliance to the EPUB3 standard
clean                     # Clean PLFA build
init                      # Setup the Git hooks (see below)
update-contributors       # Pull in new contributors from GitHub to contributors/
list                      # List all build targets
```

For completeness, the Makefile also offers the following options, but you're unlikely to need these:

```
legacy-versions           # Build legacy versions of PLFA
setup-install-bundler     # Install Ruby Bundler (needed for 'legacy-versions')
setup-install-htmlproofer # Install HTMLProofer (needed for 'test' and Git hooks)
setup-check-fix-whitespace # Check if fix-whitespace is installed (needed for Git hooks)
setup-check-epubcheck     # Check if epubcheck is installed (needed for EPUB tests)
setup-check-gem           # Check if RubyGems is installed
setup-check-npm           # Check if the Node Package Manager is installed
setup-check-stack         # Check if the Haskell Tool Stack is installed
```

The EPUB version of the book is built as part of the website, since it's hosted on the website.

Git hooks

The repository comes with several Git hooks:

1. The fix-whitespace program is run to check for whitespace violations.
2. The test suite is run to check if everything type checks.

You can install these Git hooks by calling `make init`. You can install fix-whitespace by running:

```
stack install fix-whitespace
```

If you want Stack to use your system installation of GHC, follow the instructions for Using an existing installation of GHC.

Part I

Part 1: Logical Foundations

Chapter 1

Naturals: Natural numbers

```
module plfa.part1.Naturals where
```

The night sky holds more stars than I can count, though fewer than five thousand are visible to the naked eye. The observable universe contains about seventy sextillion stars.

But the number of stars is finite, while natural numbers are infinite. Count all the stars, and you will still have as many natural numbers left over as you started with.

The naturals are an inductive datatype

Everyone is familiar with the natural numbers

```
0
1
2
3
...
```

and so on. We write `ℕ` for the *type* of natural numbers, and say that `0`, `1`, `2`, `3`, and so on are *values* of type `ℕ`, indicated by writing `0 : ℕ`, `1 : ℕ`, `2 : ℕ`, `3 : ℕ`, and so on.

The set of natural numbers is infinite, yet we can write down its definition in just a few lines. Here is the definition as a pair of inference rules:

```
--------
zero : ℕ

m : ℕ
---------
suc m : ℕ
```

And here is the definition in Agda:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

Here `ℕ` is the name of the *datatype* we are defining, and `zero` and `suc` (short for *successor*) are the *constructors* of the datatype.

Both definitions above tell us the same two things:

- *Base case*: `zero` is a natural number.
- *Inductive case*: if `m` is a natural number, then `suc m` is also a natural number.

Further, these two rules give the *only* ways of creating natural numbers. Hence, the possible natural numbers are:

```
zero
suc zero
suc (suc zero)
suc (suc (suc zero))
...
```

We write `0` as shorthand for `zero`; and `1` is shorthand for `suc zero`, the successor of zero, that is, the natural that comes after zero; and `2` is shorthand for `suc (suc zero)`, which is the same as `suc 1`, the successor of one; and `3` is shorthand for the successor of two; and so on.

Exercise `seven` (practice)

Write out `7` in longhand.

```
-- Your code goes here
```

Unpacking the inference rules

Let's unpack the inference rules. Each inference rule consists of zero or more *judgments* written above a horizontal line, called the *hypotheses*, and a single judgment written below, called the *conclusion*. The first rule is the base case. It has no hypotheses, and the conclusion asserts that `zero` is a natural. The second rule is the inductive case. It has one hypothesis, which assumes that `m` is a natural, and the conclusion asserts that `suc m` is a also a natural.

Unpacking the Agda definition

Let's unpack the Agda definition. The keyword `data` tells us this is an inductive definition, that is, that we are defining a new datatype with constructors. The phrase

```
ℕ : Set
```

tells us that `ℕ` is the name of the new datatype, and that it is a `Set`, which is the way in Agda of saying that it is a type. The keyword `where` separates the declaration of the datatype from the declaration of its constructors. Each constructor is declared on a separate line, which is indented to indicate that it belongs to the corresponding `data` declaration. The lines

```
zero : ℕ
suc  : ℕ → ℕ
```

give *signatures* specifying the types of the constructors `zero` and `suc`. They tell us that `zero` is a natural number and that `suc` takes a natural number as argument and returns a natural number.

You may have noticed that `ℕ` and `→` don't appear on your keyboard. They are symbols in *unicode*. At the end of each chapter is a list of all unicode symbols introduced in the chapter, including instructions on how to type them in the Emacs text editor. Here *type* refers to typing with fingers as opposed to data types!

The story of creation

Let's look again at the rules that define the natural numbers:

- *Base case*: `zero` is a natural number.
- *Inductive case*: if `m` is a natural number, then `suc m` is also a natural number.

Hold on! The second line defines natural numbers in terms of natural numbers. How can that possibly be allowed? Isn't this as useless a definition as "Brexit means Brexit"?

In fact, it is possible to assign our definition a meaning without resorting to unpermitted circularities. Furthermore, we can do so while only working with *finite* sets and never referring to the *infinite* set of natural numbers.

We will think of it as a creation story. To start with, we know about no natural numbers at all:

```
-- In the beginning, there are no natural numbers.
```

Now, we apply the rules to all the natural numbers we know about. The base case tells us that `zero` is a natural number, so we add it to the set of known natural numbers. The inductive case tells us that if `m` is a natural number (on the day before today) then `suc m` is also a natural number (today). We didn't know about any natural numbers before today, so the inductive case doesn't apply:

```
-- On the first day, there is one natural number.
zero : ℕ
```

Then we repeat the process. On the next day we know about all the numbers from the day before, plus any numbers added by the rules. The base case tells us that `zero` is a natural number, but we already knew that. But now the inductive case tells us that since `zero` was a natural number yesterday, then `suc zero` is a natural number today:

```
-- On the second day, there are two natural numbers.
zero : ℕ
suc zero : ℕ
```

And we repeat the process again. Now the inductive case tells us that since `zero` and `suc zero` are both natural numbers, then `suc zero` and `suc (suc zero)` are natural numbers. We already knew about the first of these, but the second is new:

```
-- On the third day, there are three natural numbers.
zero : ℕ
suc zero : ℕ
suc (suc zero) : ℕ
```

You've got the hang of it by now:

```
-- On the fourth day, there are four natural numbers.
zero : ℕ
suc zero : ℕ
suc (suc zero) : ℕ
```

```
suc (suc (suc zero)) : ℕ
```

The process continues. On the *n*'th day there will be *n* distinct natural numbers. Every natural number will appear on some given day. In particular, the number *n* first appears on day *n+1*. And we never actually define the set of numbers in terms of itself. Instead, we define the set of numbers on day *n+1* in terms of the set of numbers on day *n*.

A process like this one is called *inductive*. We start with nothing, and build up a potentially infinite set by applying rules that convert one finite set into another finite set.

The rule defining zero is called a *base case*, because it introduces a natural number even when we know no other natural numbers. The rule defining successor is called an *inductive case*, because it introduces more natural numbers once we already know some. Note the crucial role of the base case. If we only had inductive rules, then we would have no numbers in the beginning, and still no numbers on the second day, and on the third, and so on. An inductive definition lacking a base case is useless, as in the phrase "Brexit means Brexit".

Philosophy and history

A philosopher might observe that our reference to the first day, second day, and so on, implicitly involves an understanding of natural numbers. In this sense, our definition might indeed be regarded as in some sense circular, but we need not let this disturb us. Everyone possesses a good informal understanding of the natural numbers, which we may take as a foundation for their formal description.

While the natural numbers have been understood for as long as people can count, the inductive definition of the natural numbers is relatively recent. It can be traced back to Richard Dedekind's paper "*Was sind und was sollen die Zahlen?*" (What are and what should be the numbers?), published in 1888, and Giuseppe Peano's book "*Arithmetices principia, nova methodo exposita*" (The principles of arithmetic presented by a new method), published the following year.

A pragma

In Agda, any text following `--` or enclosed between `{-` and `-}` is considered a *comment*. Comments have no effect on the code, with the exception of one special kind of comment, called a *pragma*, which is enclosed between `{-#` and `#-}`.

Including the line

```
{-# BUILTIN NATURAL ℕ #-}
```

tells Agda that `ℕ` corresponds to the natural numbers, and hence one is permitted to type `0` as shorthand for `zero`, `1` as shorthand for `suc zero`, `2` as shorthand for `suc (suc zero)`, and so on. The pragma must be given a previously declared type (in this case `ℕ`) with precisely two constructors, one with no arguments (in this case `zero`), and one with a single argument of the given type (in this case `suc`).

As well as enabling the above shorthand, the pragma also enables a more efficient internal representation of naturals using the Haskell type for arbitrary-precision integers. Representing the natural *n* with `zero` and `suc` requires space proportional to *n*, whereas representing it as an arbitrary-precision integer in Haskell only requires space proportional to the logarithm of *n*.

Imports

Shortly we will want to write some equations that hold between terms involving natural numbers. To support doing so, we import the definition of equality and notations for reasoning about it from the Agda standard library:

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; _∎)
```

The first line brings the standard library module that defines equality into scope and gives it the name `Eq`. The second line opens that module, that is, adds all the names specified in the `using` clause into the current scope. In this case the names added are `_≡_`, the equality operator, and `refl`, the name for evidence that two terms are equal. The third line takes a module that specifies operators to support reasoning about equivalence, and adds all the names specified in the `using` clause into the current scope. In this case, the names added are `begin_`, `_≡⟨⟩_`, and `_∎`. We will see how these are used below. We take these as givens for now, but will see how they are defined in Chapter Equality.

Agda uses underbars to indicate where terms appear in infix or mixfix operators. Thus, `_≡_` and `_≡⟨⟩_` are infix (each operator is written between two terms), while `begin_` is prefix (it is written before a term), and `_∎` is postfix (it is written after a term).

Parentheses and semicolons are among the few characters that cannot appear in names, so we do not need extra spaces in the `using` list.

Operations on naturals are recursive functions

Now that we have the natural numbers, what can we do with them? For instance, can we define arithmetic operations such as addition and multiplication?

As a child I spent much time memorising tables of addition and multiplication. At first the rules seemed tricky and I would often make mistakes. It came as a shock to me to discover *recursion*, a simple technique by which every one of the infinite possible instances of addition and multiplication can be specified in just a couple of lines.

Here is the definition of addition in Agda:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
```

Let's unpack this definition. Addition is an infix operator. It is written with underbars where the arguments go, hence its name is `_+_`. The first line is a signature specifying the type of the operator. The type `ℕ → ℕ → ℕ`, indicates that addition accepts two naturals and returns a natural. Infix notation is just a shorthand for application; the terms `m + n` and `_+_ m n` are equivalent.

The definition has a base case and an inductive case, corresponding to those for the natural numbers. The base case says that adding zero to a number, `zero + n`, returns that number, `n`. The inductive case says that adding the successor of a number to another number, `(suc m) + n`, returns the successor of adding the two numbers, `suc (m + n)`. We say we use *pattern matching* when constructors appear on the left-hand side of an equation.

If we write `zero` as `0` and `suc m` as `1 + m`, the definition turns into two familiar equations:

```
0       + n  ≡  n
(1 + m) + n  ≡  1 + (m + n)
```

The first follows because zero is an identity for addition, and the second because addition is associative. In its most general form, associativity is written

```
(m + n) + p  ≡  m + (n + p)
```

meaning that the location of parentheses is irrelevant. We get the second equation from the third by taking `m` to be `1`, `n` to be `m`, and `p` to be `n`. We write `=` for definitions, while we write `≡` for assertions that two already defined things are the same.

The definition is *recursive*, in that the last line defines addition in terms of addition. As with the inductive definition of the naturals, the apparent circularity is not a problem. It works because addition of larger numbers is defined in terms of addition of smaller numbers. Such a definition is called *well founded*.

For example, let's add two and three:

```
_ : 2 + 3 ≡ 5
_ =
  begin
    2 + 3
  ≡⟨⟩ -- is shorthand for
    (suc (suc zero)) + (suc (suc (suc zero)))
  ≡⟨⟩ -- inductive case
    suc ((suc zero) + (suc (suc (suc zero))))
  ≡⟨⟩ -- inductive case
    suc (suc (zero + (suc (suc (suc zero)))))
  ≡⟨⟩ -- base case
    suc (suc (suc (suc (suc zero))))
  ≡⟨⟩ -- is longhand for
    5
  ∎
```

We can write the same derivation more compactly by only expanding shorthand as needed:

```
_ : 2 + 3 ≡ 5
_ =
  begin
    2 + 3
  ≡⟨⟩
    suc (1 + 3)
  ≡⟨⟩
    suc (suc (0 + 3))
  ≡⟨⟩
    suc (suc 3)
  ≡⟨⟩
    5
  ∎
```

The first line matches the inductive case by taking `m = 1` and `n = 3`, the second line matches the inductive case by taking `m = 0` and `n = 3`, and the third line matches the base case by taking `n = 3`.

Both derivations consist of a signature (written with a colon, `:`), giving a type, and a binding (written with an equal sign, `=`), giving a term of the given type. Here we use the dummy name

`_`. The dummy name can be reused, and is convenient for examples. Names other than `_` must be used only once in a module.

Here the type is `2 + 3 ≡ 5` and the term provides *evidence* for the corresponding equation, here written in tabular form as a chain of equations. The chain starts with `begin` and finishes with `∎` (pronounced "qed" or "tombstone", the latter from its appearance), and consists of a series of terms separated by `≡⟨⟩`.

In fact, both proofs are longer than need be, and Agda is satisfied with the following:

```
_ : 2 + 3 ≡ 5
_ = refl
```

Agda knows how to compute the value of `2 + 3`, and so can immediately check it is the same as `5`. A binary relation is said to be *reflexive* if every value relates to itself. Evidence that a value is equal to itself is written `refl`.

In the chains of equations, all Agda checks is that each term simplifies to the same value. If we jumble the equations, omit lines, or add extraneous lines it will still be accepted. It's up to us to write the equations in an order that makes sense to the reader.

Here `2 + 3 ≡ 5` is a type, and the chains of equations (and also `refl`) are terms of the given type; alternatively, one can think of each term as *evidence* for the assertion `2 + 3 ≡ 5`. This duality of interpretation—of a type as a proposition, and of a term as evidence—is central to how we formalise concepts in Agda, and will be a running theme throughout this book.

Note that when we use the word *evidence* it is nothing equivocal. It is not like testimony in a court which must be weighed to determine whether the witness is trustworthy. Rather, it is ironclad. The other word for evidence, which we will use interchangeably, is *proof*.

Exercise `+-example` (practice)

Compute `3 + 4`, writing out your reasoning as a chain of equations, using the equations for `+`.

```
-- Your code goes here
```

Multiplication

Once we have defined addition, we can define multiplication as repeated addition:

```
_*_ : ℕ → ℕ → ℕ
zero * n    = zero
(suc m) * n = n + (m * n)
```

Computing `m * n` returns the sum of `m` copies of `n`.

Again, rewriting turns the definition into two familiar equations:

```
0       * n  ≡  0
(1 + m) * n  ≡  n + (m * n)
```

The first follows because zero times anything is zero, and the second follows because multiplication distributes over addition. In its most general form, distribution of multiplication over addition is written

```
(m + n) * p  ≡  (m * p) + (n * p)
```

We get the second equation from the third by taking `m` to be `1`, `n` to be `m`, and `p` to be `n`, and then using the fact that one is an identity for multiplication, so `1 * n ≡ n`.

Again, the definition is well founded in that multiplication of larger numbers is defined in terms of multiplication of smaller numbers.

For example, let's multiply two and three:

```
_ =
  begin
    2 * 3
  ≡⟨⟩ -- inductive case
    3 + (1 * 3)
  ≡⟨⟩ -- inductive case
    3 + (3 + (0 * 3))
  ≡⟨⟩ -- base case
    3 + (3 + 0)
  ≡⟨⟩ -- simplify
    6
  ∎
```

The first line matches the inductive case by taking `m = 1` and `n = 3`, The second line matches the inductive case by taking `m = 0` and `n = 3`, and the third line matches the base case by taking `n = 3`. Here we have omitted the signature declaring `_ : 2 * 3 ≡ 6`, since it can easily be inferred from the corresponding term.

Exercise `*-example` (practice)

Compute `3 * 4`, writing out your reasoning as a chain of equations, using the equations for `*`. (You do not need to step through the evaluation of `+`.)

```
-- Your code goes here
```

Exercise `_^_` (recommended)

Define exponentiation, which is given by the following equations:

```
m ^ 0       =  1
m ^ (1 + n)  =  m * (m ^ n)
```

Check that `3 ^ 4` is `81`.

```
-- Your code goes here
```

Monus

We can also define subtraction. Since there are no negative natural numbers, if we subtract a larger number from a smaller number we will take the result to be zero. This adaption of subtraction to naturals is called *monus* (a twist on *minus*).

Monus is our first use of a definition that uses pattern matching against both arguments:

```
_∸_ : ℕ → ℕ → ℕ
m     ∸ zero  = m
zero  ∸ suc n = zero
suc m ∸ suc n = m ∸ n
```

We can do a simple analysis to show that all the cases are covered.

- Consider the second argument.
  - If it is `zero`, then the first equation applies.
  - If it is `suc n`, then consider the first argument.
    - If it is `zero`, then the second equation applies.
    - If it is `suc m`, then the third equation applies.

Again, the recursive definition is well founded because monus on bigger numbers is defined in terms of monus on smaller numbers.

For example, let's subtract two from three:

```
_ =
  begin
    3 ∸ 2
  ≡⟨⟩
    2 ∸ 1
  ≡⟨⟩
    1 ∸ 0
  ≡⟨⟩
    1
  ∎
```

We did not use the second equation at all, but it will be required if we try to subtract a larger number from a smaller one:

```
_ =
  begin
    2 ∸ 3
  ≡⟨⟩
    1 ∸ 2
  ≡⟨⟩
    0 ∸ 1
  ≡⟨⟩
    0
  ∎
```

Exercise `∸-example₁` and `∸-example₂` (recommended)

Compute `5 ∸ 3` and `3 ∸ 5`, writing out your reasoning as a chain of equations.

```
-- Your code goes here
```

Precedence

We often use *precedence* to avoid writing too many parentheses. Application *binds more tightly than* (or *has precedence over*) any operator, and so we may write `suc m + n` to mean `(suc m) + n`. As another example, we say that multiplication binds more tightly than addition, and so write `n + m * n` to mean `n + (m * n)`. We also sometimes say that addition *associates to the left*, and so write `m + n + p` to mean `(m + n) + p`.

In Agda the precedence and associativity of infix operators needs to be declared:

```
infixl 6 _+_ _∸_
infixl 7 _*_
```

This states operators `_+_` and `_∸_` have precedence level 6, and operator `_*_` has precedence level 7. Addition and monus bind less tightly than multiplication because they have lower precedence. Writing `infixl` indicates that all three operators associate to the left. One can also write `infixr` to indicate that an operator associates to the right, or just `infix` to indicate that parentheses are always required to disambiguate.

Currying

We have chosen to represent a function of two arguments in terms of a function of the first argument that returns a function of the second argument. This trick goes by the name *currying*.

Agda, like other functional languages such as Haskell and ML, is designed to make currying easy to use. Function arrows associate to the right and application associates to the left

`ℕ → ℕ → ℕ` stands for `ℕ → (ℕ → ℕ)`

and

`_+_ 2 3` stands for `(_+_ 2) 3`.

The term `_+_ 2` by itself stands for the function that adds two to its argument, hence applying it to three yields five.

Currying is named for Haskell Curry, after whom the programming language Haskell is also named. Curry's work dates to the 1930's. When I first learned about currying, I was told it was misattributed, since the same idea was previously proposed by Moses Schönfinkel in the 1920's. I was told a joke: "It should be called schönfinkeling, but currying is tastier". Only later did I learn that the explanation of the misattribution was itself a misattribution. The idea actually appears in the *Begriffsschrift* of Gottlob Frege, published in 1879.

The story of creation, revisited

Just as our inductive definition defines the naturals in terms of the naturals, so does our recursive definition define addition in terms of addition.

Again, it is possible to assign our definition a meaning without resorting to unpermitted circularities. We do so by reducing our definition to equivalent inference rules for judgments about equality:

```
n : ℕ
--------------
zero + n  =  n

m + n  =  p
---------------------
(suc m) + n  =  suc p
```

Here we assume we have already defined the infinite set of natural numbers, specifying the meaning of the judgment `n : ℕ`. The first inference rule is the base case. It asserts that if `n` is a natural number then adding zero to it gives `n`. The second inference rule is the inductive case. It asserts that if adding `m` and `n` gives `p`, then adding `suc m` and `n` gives `suc p`.

Again we resort to a creation story, where this time we are concerned with judgments about addition:

```
-- In the beginning, we know nothing about addition.
```

Now, we apply the rules to all the judgment we know about. The base case tells us that `zero + n = n` for every natural `n`, so we add all those equations. The inductive case tells us that if `m + n = p` (on the day before today) then `suc m + n = suc p` (today). We didn't know any equations about addition before today, so that rule doesn't give us any new equations:

```
-- On the first day, we know about addition of 0.
0 + 0 = 0     0 + 1 = 1    0 + 2 = 2     ...
```

Then we repeat the process, so on the next day we know about all the equations from the day before, plus any equations added by the rules. The base case tells us nothing new, but now the inductive case adds more equations:

```
-- On the second day, we know about addition of 0 and 1.
0 + 0 = 0     0 + 1 = 1     0 + 2 = 2     0 + 3 = 3     ...
1 + 0 = 1     1 + 1 = 2     1 + 2 = 3     1 + 3 = 4     ...
```

And we repeat the process again:

```
-- On the third day, we know about addition of 0, 1, and 2.
0 + 0 = 0     0 + 1 = 1     0 + 2 = 2     0 + 3 = 3     ...
1 + 0 = 1     1 + 1 = 2     1 + 2 = 3     1 + 3 = 4     ...
2 + 0 = 2     2 + 1 = 3     2 + 2 = 4     2 + 3 = 5     ...
```

You've got the hang of it by now:

```
-- On the fourth day, we know about addition of 0, 1, 2, and 3.
0 + 0 = 0     0 + 1 = 1     0 + 2 = 2     0 + 3 = 3     ...
1 + 0 = 1     1 + 1 = 2     1 + 2 = 3     1 + 3 = 4     ...
2 + 0 = 2     2 + 1 = 3     2 + 2 = 4     2 + 3 = 5     ...
3 + 0 = 3     3 + 1 = 4     3 + 2 = 5     3 + 3 = 6     ...
```

The process continues. On the *m*'th day we will know all the equations where the first number is less than *m*.

As we can see, the reasoning that justifies inductive and recursive definitions is quite similar. They might be considered two sides of the same coin.

The story of creation, finitely

The above story was told in a stratified way. First, we create the infinite set of naturals. We take that set as given when creating instances of addition, so even on day one we have an infinite set of instances.

Instead, we could choose to create both the naturals and the instances of addition at the same time. Then on any day there would be only a finite set of instances:

```
-- In the beginning, we know nothing.
```

Now, we apply the rules to all the judgment we know about. Only the base case for naturals applies:

```
-- On the first day, we know zero.
0 : ℕ
```

Again, we apply all the rules we know. This gives us a new natural, and our first equation about addition.

```
-- On the second day, we know one and all sums that yield zero.
0 : ℕ
1 : ℕ    0 + 0 = 0
```

Then we repeat the process. We get one more equation about addition from the base case, and also get an equation from the inductive case, applied to equation of the previous day:

```
-- On the third day, we know two and all sums that yield one.
0 : ℕ
1 : ℕ    0 + 0 = 0
2 : ℕ    0 + 1 = 1   1 + 0 = 1
```

You've got the hang of it by now:

```
-- On the fourth day, we know three and all sums that yield two.
0 : ℕ
1 : ℕ    0 + 0 = 0
2 : ℕ    0 + 1 = 1   1 + 0 = 1
3 : ℕ    0 + 2 = 2   1 + 1 = 2    2 + 0 = 2
```

On the *n*'th day there will be *n* distinct natural numbers, and $n \times (n-1) / 2$ equations about addition. The number *n* and all equations for addition of numbers less than *n* first appear by day $n+1$. This gives an entirely finitist view of infinite sets of data and equations relating the data.

Writing definitions interactively

Agda is designed to be used with the Emacs text editor, and the two in combination provide features that help to create definitions and proofs interactively.

Begin by typing:

```
_+_ : ℕ → ℕ → ℕ
m + n = ?
```

The question mark indicates that you would like Agda to help with filling in that part of the code. If you type `C-c C-l` (pressing the control key while hitting the `c` key followed by the `l` key), which stands for **l**oad, the question mark will be replaced:

```
_+_ : ℕ → ℕ → ℕ
m + n = { }0
```

The empty braces are called a *hole*, and 0 is a number used for referring to the hole. The hole will display highlighted in green. Emacs will also create a window displaying the text

```
?0 : ℕ
```

to indicate that hole 0 is to be filled in with a term of type `ℕ`. Typing `C-c C-f` (for **f**orward) will move you into the next hole.

We wish to define addition by recursion on the first argument. Move the cursor into the hole and type `C-c C-c` (for **c**ase). You will be given the prompt:

```
pattern variables to case (empty for split on result):
```

Typing `m` will cause a split on that variable, resulting in an update to the code:

```
_+_ : ℕ → ℕ → ℕ
zero + n = { }0
suc m + n = { }1
```

There are now two holes, and the window at the bottom tells you the required type of each:

```
?0 : ℕ
?1 : ℕ
```

Going into hole 0 and type `C-c C-,` will display information on the required type of the hole, and what free variables are available:

```
Goal: ℕ
————————————————————————————————————————————————————————————
n : ℕ
```

This strongly suggests filling the hole with `n`. After the hole is filled, you can type `C-c C-space`, which will remove the hole:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = { }1
```

Again, going into hole 1 and type `C-c C-,` will display information on the required type of the hole, and what free variables are available:

```
Goal: ℕ
————————————————————————————————————————————————————————————
n : ℕ
m : ℕ
```

Going into the hole and type `C-c C-r` (for **r**efine) will fill it in with a constructor (if there is a unique choice) or tell you what constructors you might use, if there is a choice. In this case, it displays the following:

```
Don't know which constructor to introduce of zero or suc
```

Filling the hole with `suc ?` and typing `C-c C-space` results in the following:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc { }1
```

Going into the new hole and typing `C-c C-,` gives similar information to before:

```
Goal: ℕ
————————————————————————————————————————————————————————————
n : ℕ
m : ℕ
```

We can fill the hole with `m + n` and type `C-c C-space` to complete the program:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

Exploiting interaction to this degree is probably not helpful for a program this simple, but the same techniques can help with more complex programs. Even for a program this simple, using `C-c C-c` to split cases can be helpful.

More pragmas

Including the lines

```
{-# BUILTIN NATPLUS _+_ #-}
{-# BUILTIN NATTIMES _*_ #-}
{-# BUILTIN NATMINUS _∸_ #-}
```

tells Agda that these three operators correspond to the usual ones, and enables it to perform these computations using the corresponding Haskell operators on the arbitrary-precision integer type. Representing naturals with `zero` and `suc` requires time proportional to m to add m and n, whereas representing naturals as integers in Haskell requires time proportional to the larger of the logarithms of m and n. Similarly, representing naturals with `zero` and `suc` requires time proportional to the product of m and n to multiply m and n, whereas representing naturals as integers in Haskell requires time proportional to the sum of the logarithms of m and n.

Exercise `Bin` (stretch)

A more efficient representation of natural numbers uses a binary rather than a unary system. We represent a number as a bitstring:

```
data Bin : Set where
  ⟨⟩ : Bin
  _O : Bin → Bin
  _I : Bin → Bin
```

For instance, the bitstring

```
1011
```

standing for the number eleven is encoded as

```
⟨⟩ I O I I
```

Representations are not unique due to leading zeros. Hence, eleven is also represented by `001011`, encoded as:

```
⟨⟩ O O I O I I
```

Define a function

```
inc : Bin → Bin
```

that converts a bitstring to the bitstring for the next higher number. For example, since `1100` encodes twelve, we should have:

```
inc (⟨⟩ I O I I) ≡ ⟨⟩ I I O O
```

Confirm that this gives the correct answer for the bitstrings encoding zero through four.

Using the above, define a pair of functions to convert between the two representations.

```
to   : ℕ → Bin
from : Bin → ℕ
```

For the former, choose the bitstring to have no leading zeros if it represents a positive natural, and represent zero by `⟨⟩ O`. Confirm that these both give the correct answer for zero through four.

```
-- Your code goes here
```

Standard library

At the end of each chapter, we will show where to find relevant definitions in the standard library. The naturals, constructors for them, and basic operators upon them, are defined in the standard library module `Data.Nat`:

```
-- import Data.Nat using (ℕ; zero; suc; _+_; _*_; _^_; _∸_)
```

Normally, we will show an import as running code, so Agda will complain if we attempt to import a definition that is not available. This time, however, we have only shown the import as a comment. Both this chapter and the standard library invoke the `NATURAL` pragma, the former on `ℕ`, and the latter on the equivalent type `Data.Nat.ℕ`. Such a pragma can only be invoked once, as invoking it twice would raise confusion as to whether `2` is a value of type `ℕ` or type `Data.Nat.ℕ`. Similar confusions arise if other pragmas are invoked twice. For this reason, we will usually avoid pragmas in future chapters. Information on pragmas can be found in the Agda documentation.

Unicode

This chapter uses the following unicode:

```
ℕ  U+2115  DOUBLE-STRUCK CAPITAL N (\bN)
→  U+2192  RIGHTWARDS ARROW (\to, \r, \->)
```

```
∸  U+2238  DOT MINUS (\.-)
≡  U+2261  IDENTICAL TO (\==)
⟨  U+27E8  MATHEMATICAL LEFT ANGLE BRACKET (\<)
⟩  U+27E9  MATHEMATICAL RIGHT ANGLE BRACKET (\>)
∎  U+220E  END OF PROOF (\qed)
```

Each line consists of the Unicode character (`ℕ`), the corresponding code point (`U+2115`), the name of the character (`DOUBLE-STRUCK CAPITAL N`), and the sequence to type into Emacs to generate the character (`\bN`).

The command `\r` gives access to a wide variety of rightward arrows. After typing `\r` , one can access the many available arrows by using the left, right, up, and down keys to navigate. The command remembers where you navigated to the last time, and starts with the same character next time. The command `\l` works similarly for left arrows. In place of left, right, up, and down keys, one may also use control characters:

```
C-b  left (backward one character)
C-f  right (forward one character)
C-p  up (to the previous line)
C-n  down (to the next line)
```

We write `C-b` to stand for control-b, and similarly. One can also navigate left and right by typing the digits that appear in the displayed list.

For a full list of supported characters, use `agda-input-show-translations` with:

```
M-x agda-input-show-translations
```

All the characters supported by `agda-mode` are shown. We write M-x to stand for typing `ESC` followed by `x` .

If you want to know how you input a specific Unicode character in an agda file, move the cursor onto the character and use `quail-show-key` with:

```
M-x quail-show-key
```

You'll see a key sequence of the character in mini buffer. If you run `M-x quail-show-key` on say `∸` , you will see `\.-` for the character.

Chapter 2

Induction: Proof by Induction

```
module plfa.part1.Induction where
```

> Induction makes you feel guilty for getting something out of nothing ... but it is one of the greatest ideas of civilization. – Herbert Wilf

Now that we've defined the naturals and operations upon them, our next step is to learn how to prove properties that they satisfy. As hinted by their name, properties of *inductive datatypes* are proved by *induction*.

Imports

We require equality as in the previous chapter, plus the naturals and some operations upon them. We also import a couple of new operations, `cong`, `sym`, and `_≡⟨_⟩_`, which are explained below:

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; sym)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _∎)
open import Data.Nat using (ℕ; zero; suc; _+_; _*_; _∸_)
```

Properties of operators

Operators pop up all the time, and mathematicians have agreed on names for some of the most common properties.

- *Identity*. Operator `+` has left identity `0` if `0 + n ≡ n`, and right identity `0` if `n + 0 ≡ n`, for all `n`. A value that is both a left and right identity is just called an identity. Identity is also sometimes called *unit*.
- *Associativity*. Operator `+` is associative if the location of parentheses does not matter: `(m + n) + p ≡ m + (n + p)`, for all `m`, `n`, and `p`.
- *Commutativity*. Operator `+` is commutative if order of arguments does not matter: `m + n ≡ n + m`, for all `m` and `n`.

- *Distributivity*. Operator `*` distributes over operator `+` from the left if `(m + n) * p ≡ (m * p) + (n * p)`, for all `m`, `n`, and `p`, and from the right if `m * (p + q) ≡ (m * p) + (m * q)`, for all `m`, `p`, and `q`.

Addition has identity `0` and multiplication has identity `1`; addition and multiplication are both associative and commutative; and multiplication distributes over addition.

If you ever bump into an operator at a party, you now know how to make small talk, by asking whether it has a unit and is associative or commutative. If you bump into two operators, you might ask them if one distributes over the other.

Less frivolously, if you ever bump into an operator while reading a technical paper, this gives you a way to orient yourself, by checking whether or not it has an identity, is associative or commutative, or distributes over another operator. A careful author will often call out these properties—or their lack—for instance by pointing out that a newly introduced operator is associative but not commutative.

Exercise `operators` (practice)

Give another example of a pair of operators that have an identity and are associative, commutative, and distribute over one another. (You do not have to prove these properties.)

Give an example of an operator that has an identity and is associative but is not commutative. (You do not have to prove these properties.)

Associativity

One property of addition is that it is *associative*, that is, that the location of the parentheses does not matter:

```
(m + n) + p ≡ m + (n + p)
```

Here `m`, `n`, and `p` are variables that range over all natural numbers.

We can test the proposition by choosing specific numbers for the three variables:

```
_ : (3 + 4) + 5 ≡ 3 + (4 + 5)
_ =
  begin
    (3 + 4) + 5
  ≡⟨⟩
    7 + 5
  ≡⟨⟩
    12
  ≡⟨⟩
    3 + 9
  ≡⟨⟩
    3 + (4 + 5)
  ∎
```

Here we have displayed the computation as a chain of equations, one term to a line. It is often easiest to read such chains from the top down until one reaches the simplest term (in this case, `12`), and then from the bottom up until one reaches the same term.

The test reveals that associativity is perhaps not as obvious as first it appears. Why should `7 + 5` be the same as `3 + 9`? We might want to gather more evidence, testing the proposition by choosing other numbers. But—since there are an infinite number of naturals—testing can never be complete. Is there any way we can be sure that associativity holds for *all* the natural numbers?

The answer is yes! We can prove a property holds for all naturals using *proof by induction*.

Proof by induction

Recall the definition of natural numbers consists of a *base case* which tells us that `zero` is a natural, and an *inductive case* which tells us that if `m` is a natural then `suc m` is also a natural.

Proof by induction follows the structure of this definition. To prove a property of natural numbers by induction, we need to prove two cases. First is the *base case*, where we show the property holds for `zero`. Second is the *inductive case*, where we assume the property holds for an arbitrary natural `m` (we call this the *inductive hypothesis*), and then show that the property must also hold for `suc m`.

If we write `P m` for a property of `m`, then what we need to demonstrate are the following two inference rules:

```
------
P zero

P m
---------
P (suc m)
```

Let's unpack these rules. The first rule is the base case, and requires we show that property `P` holds for `zero`. The second rule is the inductive case, and requires we show that if we assume the inductive hypothesis—namely that `P` holds for `m`—then it follows that `P` also holds for `suc m`.

Why does this work? Again, it can be explained by a creation story. To start with, we know no properties:

```
-- In the beginning, no properties are known.
```

Now, we apply the two rules to all the properties we know about. The base case tells us that `P zero` holds, so we add it to the set of known properties. The inductive case tells us that if `P m` holds (on the day before today) then `P (suc m)` also holds (today). We didn't know about any properties before today, so the inductive case doesn't apply:

```
-- On the first day, one property is known.
P zero
```

Then we repeat the process, so on the next day we know about all the properties from the day before, plus any properties added by the rules. The base case tells us that `P zero` holds, but we already knew that. But now the inductive case tells us that since `P zero` held yesterday, then `P (suc zero)` holds today:

```
-- On the second day, two properties are known.
P zero
P (suc zero)
```

And we repeat the process again. Now the inductive case tells us that since `P zero` and `P (suc zero)` both hold, then `P (suc zero)` and `P (suc (suc zero))` also hold. We already

knew about the first of these, but the second is new:

```
-- On the third day, three properties are known.
P zero
P (suc zero)
P (suc (suc zero))
```

You've got the hang of it by now:

```
-- On the fourth day, four properties are known.
P zero
P (suc zero)
P (suc (suc zero))
P (suc (suc (suc zero)))
```

The process continues. On the *n*'th day there will be *n* distinct properties that hold. The property of every natural number will appear on some given day. In particular, the property `P n` first appears on day *n+1*.

Our first proof: associativity

To prove associativity, we take `P m` to be the property:

```
(m + n) + p ≡ m + (n + p)
```

Here `n` and `p` are arbitrary natural numbers, so if we can show the equation holds for all `m` it will also hold for all `n` and `p`. The appropriate instances of the inference rules are:

```
------------------------------
(zero + n) + p ≡ zero + (n + p)

(m + n) + p ≡ m + (n + p)
--------------------------------
(suc m + n) + p ≡ suc m + (n + p)
```

If we can demonstrate both of these, then associativity of addition follows by induction.

Here is the proposition's statement and proof:

```
+-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p =
  begin
    (zero + n) + p
  ≡⟨⟩
    n + p
  ≡⟨⟩
    zero + (n + p)
  ∎
+-assoc (suc m) n p =
  begin
    (suc m + n) + p
  ≡⟨⟩
    suc (m + n) + p
  ≡⟨⟩
    suc ((m + n) + p)
  ≡⟨ cong suc (+-assoc m n p) ⟩
```

```
    suc (m + (n + p))
  ≡⟨ ⟩
    suc m + (n + p)
  ∎
```

We have named the proof `+-assoc`. In Agda, identifiers can consist of any sequence of characters not including spaces or the characters `@.(){};_`.

Let's unpack this code. The signature states that we are defining the identifier `+-assoc` which provides evidence for the proposition:

```
∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
```

The upside down A is pronounced "for all", and the proposition asserts that for all natural numbers `m`, `n`, and `p` the equation `(m + n) + p ≡ m + (n + p)` holds. Evidence for the proposition is a function that accepts three natural numbers, binds them to `m`, `n`, and `p`, and returns evidence for the corresponding instance of the equation.

For the base case, we must show:

```
(zero + n) + p ≡ zero + (n + p)
```

Simplifying both sides with the base case of addition yields the equation:

```
n + p ≡ n + p
```

This holds trivially. Reading the chain of equations in the base case of the proof, the top and bottom of the chain match the two sides of the equation to be shown, and reading down from the top and up from the bottom takes us to `n + p` in the middle. No justification other than simplification is required.

For the inductive case, we must show:

```
(suc m + n) + p ≡ suc m + (n + p)
```

Simplifying both sides with the inductive case of addition yields the equation:

```
suc ((m + n) + p) ≡ suc (m + (n + p))
```

This in turn follows by prefacing `suc` to both sides of the induction hypothesis:

```
(m + n) + p ≡ m + (n + p)
```

Reading the chain of equations in the inductive case of the proof, the top and bottom of the chain match the two sides of the equation to be shown, and reading down from the top and up from the bottom takes us to the simplified equation above. The remaining equation does not follow from simplification alone, so we use an additional operator for chain reasoning, `_≡⟨_⟩_`, where a justification for the equation appears within angle brackets. The justification given is:

```
⟨ cong suc (+-assoc m n p) ⟩
```

Here, the recursive invocation `+-assoc m n p` has as its type the induction hypothesis, and `cong suc` prefaces `suc` to each side to yield the needed equation.

A relation is said to be a *congruence* for a given function if it is preserved by applying that function. If `e` is evidence that `x ≡ y`, then `cong f e` is evidence that `f x ≡ f y`, for any function `f`.

Here the inductive hypothesis is not assumed, but instead proved by a recursive invocation of the function we are defining, `+-assoc m n p`. As with addition, this is well founded because associativity of larger numbers is proved in terms of associativity of smaller numbers. In this case, `assoc (suc m) n p` is proved using `assoc m n p`. The correspondence between proof by induction and definition by recursion is one of the most appealing aspects of Agda.

Induction as recursion

As a concrete example of how induction corresponds to recursion, here is the computation that occurs when instantiating `m` to `2` in the proof of associativity.

```
+-assoc-2 : ∀ (n p : ℕ) → (2 + n) + p ≡ 2 + (n + p)
+-assoc-2 n p =
  begin
    (2 + n) + p
  ≡⟨⟩
    suc (1 + n) + p
  ≡⟨⟩
    suc ((1 + n) + p)
  ≡⟨ cong suc (+-assoc-1 n p) ⟩
    suc (1 + (n + p))
  ≡⟨⟩
    2 + (n + p)
  ∎
  where
  +-assoc-1 : ∀ (n p : ℕ) → (1 + n) + p ≡ 1 + (n + p)
  +-assoc-1 n p =
    begin
      (1 + n) + p
    ≡⟨⟩
      suc (0 + n) + p
    ≡⟨⟩
      suc ((0 + n) + p)
    ≡⟨ cong suc (+-assoc-0 n p) ⟩
      suc (0 + (n + p))
    ≡⟨⟩
      1 + (n + p)
    ∎
    where
    +-assoc-0 : ∀ (n p : ℕ) → (0 + n) + p ≡ 0 + (n + p)
    +-assoc-0 n p =
      begin
        (0 + n) + p
      ≡⟨⟩
        n + p
      ≡⟨⟩
        0 + (n + p)
      ∎
```

Terminology and notation

The symbol `∀` appears in the statement of associativity to indicate that it holds for all numbers `m`, `n`, and `p`. We refer to `∀` as the *universal quantifier*, and it is discussed further in Chapter

Quantifiers.

Evidence for a universal quantifier is a function. The notations

```
+-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
```

and

```
+-assoc : ∀ (m : ℕ) → ∀ (n : ℕ) → ∀ (p : ℕ) → (m + n) + p ≡ m + (n + p)
```

are equivalent. They differ from a function type such as `ℕ → ℕ → ℕ` in that variables are associated with each argument type, and the result type may mention (or depend upon) these variables; hence they are called *dependent functions*.

Our second proof: commutativity

Another important property of addition is that it is *commutative*, that is, that the order of the operands does not matter:

```
m + n ≡ n + m
```

The proof requires that we first demonstrate two lemmas.

The first lemma

The base case of the definition of addition states that zero is a left-identity:

```
zero + n ≡ n
```

Our first lemma states that zero is also a right-identity:

```
m + zero ≡ m
```

Here is the lemma's statement and proof:

```
+-identityʳ : ∀ (m : ℕ) → m + zero ≡ m
+-identityʳ zero =
  begin
    zero + zero
  ≡⟨⟩
    zero
  ∎
+-identityʳ (suc m) =
  begin
    suc m + zero
  ≡⟨⟩
    suc (m + zero)
  ≡⟨ cong suc (+-identityʳ m) ⟩
    suc m
  ∎
```

The signature states that we are defining the identifier `+-identityʳ` which provides evidence for the proposition:

```
∀ (m : ℕ) → m + zero ≡ m
```

Evidence for the proposition is a function that accepts a natural number, binds it to `m`, and returns evidence for the corresponding instance of the equation. The proof is by induction on `m`.

For the base case, we must show:

```
zero + zero ≡ zero
```

Simplifying with the base case of addition, this is straightforward.

For the inductive case, we must show:

```
(suc m) + zero = suc m
```

Simplifying both sides with the inductive case of addition yields the equation:

```
suc (m + zero) = suc m
```

This in turn follows by prefacing `suc` to both sides of the induction hypothesis:

```
m + zero ≡ m
```

Reading the chain of equations down from the top and up from the bottom takes us to the simplified equation above. The remaining equation has the justification:

```
⟨ cong suc (+-identityʳ m) ⟩
```

Here, the recursive invocation `+-identityʳ m` has as its type the induction hypothesis, and `cong suc` prefaces `suc` to each side to yield the needed equation. This completes the first lemma.

The second lemma

The inductive case of the definition of addition pushes `suc` on the first argument to the outside:

```
suc m + n ≡ suc (m + n)
```

Our second lemma does the same for `suc` on the second argument:

```
m + suc n ≡ suc (m + n)
```

Here is the lemma's statement and proof:

```
+-suc : ∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
+-suc zero n =
  begin
    zero + suc n
  ≡⟨⟩
    suc n
  ≡⟨⟩
    suc (zero + n)
  ∎
+-suc (suc m) n =
  begin
    suc m + suc n
```

```
  ≡⟨⟩
    suc (m + suc n)
  ≡⟨ cong suc (+-suc m n) ⟩
    suc (suc (m + n))
  ≡⟨⟩
    suc (suc m + n)
  ∎
```

The signature states that we are defining the identifier `+-suc` which provides evidence for the proposition:

```
∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
```

Evidence for the proposition is a function that accepts two natural numbers, binds them to `m` and `n`, and returns evidence for the corresponding instance of the equation. The proof is by induction on `m`.

For the base case, we must show:

```
zero + suc n ≡ suc (zero + n)
```

Simplifying with the base case of addition, this is straightforward.

For the inductive case, we must show:

```
suc m + suc n ≡ suc (suc m + n)
```

Simplifying both sides with the inductive case of addition yields the equation:

```
suc (m + suc n) ≡ suc (suc (m + n))
```

This in turn follows by prefacing `suc` to both sides of the induction hypothesis:

```
m + suc n ≡ suc (m + n)
```

Reading the chain of equations down from the top and up from the bottom takes us to the simplified equation in the middle. The remaining equation has the justification:

```
⟨ cong suc (+-suc m n) ⟩
```

Here, the recursive invocation `+-suc m n` has as its type the induction hypothesis, and `cong suc` prefaces `suc` to each side to yield the needed equation. This completes the second lemma.

The proposition

Finally, here is our proposition's statement and proof:

```
+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
+-comm m zero =
  begin
    m + zero
  ≡⟨ +-identityʳ m ⟩
    m
  ≡⟨⟩
    zero + m
```

```
  ∎
+-comm m (suc n) =
  begin
    m + suc n
  ≡⟨ +-suc m n ⟩
    suc (m + n)
  ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
  ≡⟨⟩
    suc n + m
  ∎
```

The first line states that we are defining the identifier `+-comm` which provides evidence for the proposition:

```
∀ (m n : ℕ) → m + n ≡ n + m
```

Evidence for the proposition is a function that accepts two natural numbers, binds them to `m` and `n`, and returns evidence for the corresponding instance of the equation. The proof is by induction on `n`. (Not on `m` this time!)

For the base case, we must show:

```
m + zero ≡ zero + m
```

Simplifying both sides with the base case of addition yields the equation:

```
m + zero ≡ m
```

The remaining equation has the justification `⟨ +-identityʳ m ⟩`, which invokes the first lemma.

For the inductive case, we must show:

```
m + suc n ≡ suc n + m
```

Simplifying both sides with the inductive case of addition yields the equation:

```
m + suc n ≡ suc (n + m)
```

We show this in two steps. First, we have:

```
m + suc n ≡ suc (m + n)
```

which is justified by the second lemma, `⟨ +-suc m n ⟩`. Then we have

```
suc (m + n) ≡ suc (n + m)
```

which is justified by congruence and the induction hypothesis, `⟨ cong suc (+-comm m n) ⟩`. This completes the proof.

Agda requires that identifiers are defined before they are used, so we must present the lemmas before the main proposition, as we have done above. In practice, one will often attempt to prove the main proposition first, and the equations required to do so will suggest what lemmas to prove.

Our first corollary: rearranging

We can apply associativity to rearrange parentheses however we like. Here is an example:

```
+-rearrange : ∀ (m n p q : ℕ) → (m + n) + (p + q) ≡ m + (n + p) + q
+-rearrange m n p q =
  begin
    (m + n) + (p + q)
  ≡⟨ +-assoc m n (p + q) ⟩
    m + (n + (p + q))
  ≡⟨ cong (m +_) (sym (+-assoc n p q)) ⟩
    m + ((n + p) + q)
  ≡⟨ sym (+-assoc m (n + p) q) ⟩
    (m + (n + p)) + q
  ∎
```

No induction is required, we simply apply associativity twice. A few points are worthy of note.

First, addition associates to the left, so `m + (n + p) + q` stands for `(m + (n + p)) + q`.

Second, we use `sym` to interchange the sides of an equation. Proposition `+-assoc n p q` shifts parentheses from right to left:

```
(n + p) + q ≡ n + (p + q)
```

To shift them the other way, we use `sym (+-assoc n p q)`:

```
n + (p + q) ≡ (n + p) + q
```

In general, if `e` provides evidence for `x ≡ y` then `sym e` provides evidence for `y ≡ x`.

Third, Agda supports a variant of the *section* notation introduced by Richard Bird. We write `(x +_)` for the function that applied to `y` returns `x + y`. Thus, applying the congruence `cong (m +_)` takes the above equation into:

```
m + (n + (p + q)) ≡ m + ((n + p) + q)
```

Similarly, we write `(_+ x)` for the function that applied to `y` returns `y + x`; the same works for any infix operator.

Creation, one last time

Returning to the proof of associativity, it may be helpful to view the inductive proof (or, equivalently, the recursive definition) as a creation story. This time we are concerned with judgments asserting associativity:

```
  -- In the beginning, we know nothing about associativity.
```

Now, we apply the rules to all the judgments we know about. The base case tells us that `(zero + n) + p ≡ zero + (n + p)` for every natural `n` and `p`. The inductive case tells us that if `(m + n) + p ≡ m + (n + p)` (on the day before today) then `(suc m + n) + p ≡ suc m + (n + p)` (today). We didn't know any judgments about associativity before today, so that rule doesn't give us any new judgments:

```
-- On the first day, we know about associativity of 0.
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
```

Then we repeat the process, so on the next day we know about all the judgments from the day before, plus any judgments added by the rules. The base case tells us nothing new, but now the inductive case adds more judgments:

```
-- On the second day, we know about associativity of 0 and 1.
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
(1 + 0) + 0 ≡ 1 + (0 + 0)   ...   (1 + 4) + 5 ≡ 1 + (4 + 5)   ...
```

And we repeat the process again:

```
-- On the third day, we know about associativity of 0, 1, and 2.
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
(1 + 0) + 0 ≡ 1 + (0 + 0)   ...   (1 + 4) + 5 ≡ 1 + (4 + 5)   ...
(2 + 0) + 0 ≡ 2 + (0 + 0)   ...   (2 + 4) + 5 ≡ 2 + (4 + 5)   ...
```

You've got the hang of it by now:

```
-- On the fourth day, we know about associativity of 0, 1, 2, and 3.
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
(1 + 0) + 0 ≡ 1 + (0 + 0)   ...   (1 + 4) + 5 ≡ 1 + (4 + 5)   ...
(2 + 0) + 0 ≡ 2 + (0 + 0)   ...   (2 + 4) + 5 ≡ 2 + (4 + 5)   ...
(3 + 0) + 0 ≡ 3 + (0 + 0)   ...   (3 + 4) + 5 ≡ 3 + (4 + 5)   ...
```

The process continues. On the *m*'th day we will know all the judgments where the first number is less than *m*.

There is also a completely finite approach to generating the same equations, which is left as an exercise for the reader.

Exercise `finite-|-assoc` (stretch)

Write out what is known about associativity of addition on each of the first four days using a finite story of creation, as earlier.

```
-- Your code goes here
```

Associativity with rewrite

There is more than one way to skin a cat. Here is a second proof of associativity of addition in Agda, using `rewrite` rather than chains of equations:

```
+-assoc′ : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc′ zero n p                        = refl
+-assoc′ (suc m) n p rewrite +-assoc′ m n p = refl
```

For the base case, we must show:

```
(zero + n) + p ≡ zero + (n + p)
```

Simplifying both sides with the base case of addition yields the equation:

```
n + p ≡ n + p
```

This holds trivially. The proof that a term is equal to itself is written `refl`.

For the inductive case, we must show:

```
(suc m + n) + p ≡ suc m + (n + p)
```

Simplifying both sides with the inductive case of addition yields the equation:

```
suc ((m + n) + p) ≡ suc (m + (n + p))
```

After rewriting with the inductive hypothesis these two terms are equal, and the proof is again given by `refl`. Rewriting by a given equation is indicated by the keyword `rewrite` followed by a proof of that equation. Rewriting avoids not only chains of equations but also the need to invoke `cong`.

Commutativity with rewrite

Here is a second proof of commutativity of addition, using `rewrite` rather than chains of equations:

```
+-identity′ : ∀ (n : ℕ) → n + zero ≡ n
+-identity′ zero = refl
+-identity′ (suc n) rewrite +-identity′ n = refl

+-suc′ : ∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
+-suc′ zero n = refl
+-suc′ (suc m) n rewrite +-suc′ m n = refl

+-comm′ : ∀ (m n : ℕ) → m + n ≡ n + m
+-comm′ m zero rewrite +-identity′ m = refl
+-comm′ m (suc n) rewrite +-suc′ m n | +-comm′ m n = refl
```

In the final line, rewriting with two equations is indicated by separating the two proofs of the relevant equations by a vertical bar; the rewrite on the left is performed before that on the right.

Building proofs interactively

It is instructive to see how to build the alternative proof of associativity using the interactive features of Agda in Emacs. Begin by typing:

```
+-assoc′ : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc′ m n p = ?
```

The question mark indicates that you would like Agda to help with filling in that part of the code. If you type `C-c C-l` (control-c followed by control-l), the question mark will be replaced:

```
+-assoc′ : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc′ m n p = { }0
```

The empty braces are called a *hole*, and 0 is a number used for referring to the hole. The hole may display highlighted in green. Emacs will also create a new window at the bottom of the screen displaying the text:

```
?0 : ((m + n) + p) ≡ (m + (n + p))
```

This indicates that hole 0 is to be filled in with a proof of the stated judgment.

We wish to prove the proposition by induction on `m`. Move the cursor into the hole and type `C-c C-c`. You will be given the prompt:

```
pattern variables to case (empty for split on result):
```

Typing `m` will cause a split on that variable, resulting in an update to the code:

```
+-assoc′ : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc′ zero n p = { }0
+-assoc′ (suc m) n p = { }1
```

There are now two holes, and the window at the bottom tells you what each is required to prove:

```
?0 : ((zero + n) + p) ≡ (zero + (n + p))
?1 : ((suc m + n) + p) ≡ (suc m + (n + p))
```

Going into hole 0 and typing `C-c C-,` will display the text:

```
Goal: (n + p) ≡ (n + p)
————————————————————————————————————————————————————————————
p : ℕ
n : ℕ
```

This indicates that after simplification the goal for hole 0 is as stated, and that variables `p` and `n` of the stated types are available to use in the proof. The proof of the given goal is trivial, and going into the goal and typing `C-c C-r` will fill it in. Typing `C-c C-l` renumbers the remaining hole to 0:

```
+-assoc′ : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc′ zero n p = refl
+-assoc′ (suc m) n p = { }0
```

Going into the new hole 0 and typing `C-c C-,` will display the text:

```
Goal: suc ((m + n) + p) ≡ suc (m + (n + p))
————————————————————————————————————————————————————————————
p : ℕ
n : ℕ
m : ℕ
```

Again, this gives the simplified goal and the available variables. In this case, we need to rewrite by the induction hypothesis, so let's edit the text accordingly:

```
+-assoc′ : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc′ zero n p = refl
+-assoc′ (suc m) n p rewrite +-assoc′ m n p = { }0
```

Going into the remaining hole and typing `C-c C-,` will display the text:

```
Goal: suc (m + (n + p)) ≡ suc (m + (n + p))
————————————————————————————————————————————
p : ℕ
n : ℕ
m : ℕ
```

The proof of the given goal is trivial, and going into the goal and typing `C-c C-r` will fill it in, completing the proof:

```
+-assoc′ : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc′ zero n p = refl
+-assoc′ (suc m) n p rewrite +-assoc′ m n p = refl
```

Exercise `+-swap` (recommended)

Show

```
m + (n + p) ≡ n + (m + p)
```

for all naturals `m`, `n`, and `p`. No induction is needed, just apply the previous results which show addition is associative and commutative.

```
-- Your code goes here
```

Exercise `*-distrib-+` (recommended)

Show multiplication distributes over addition, that is,

```
(m + n) * p ≡ m * p + n * p
```

for all naturals `m`, `n`, and `p`.

```
-- Your code goes here
```

Exercise `*-assoc` (recommended)

Show multiplication is associative, that is,

```
(m * n) * p ≡ m * (n * p)
```

for all naturals `m`, `n`, and `p`.

```
-- Your code goes here
```

Exercise `*-comm` (practice)

Show multiplication is commutative, that is,

```
m * n ≡ n * m
```

for all naturals `m` and `n`. As with commutativity of addition, you will need to formulate and prove suitable lemmas.

```
-- Your code goes here
```

Exercise `0∸n≡0` (practice)

Show

```
zero ∸ n ≡ zero
```

for all naturals `n`. Did your proof require induction?

```
-- Your code goes here
```

Exercise `∸-|-assoc` (practice)

Show that monus associates with addition, that is,

```
m ∸ n ∸ p ≡ m ∸ (n + p)
```

for all naturals `m`, `n`, and `p`.

```
-- Your code goes here
```

Exercise `+*^` (stretch)

Show the following three laws

```
m ^ (n + p) ≡ (m ^ n) * (m ^ p)  (^-distribˡ-|-*)
(m * n) ^ p ≡ (m ^ p) * (n ^ p)  (^-distribʳ-*)
(m ^ n) ^ p ≡ m ^ (n * p)        (^-*-assoc)
```

for all `m`, `n`, and `p`.

Exercise `Bin-laws` (stretch)

Recall that Exercise Bin defines a datatype `Bin` of bitstrings representing natural numbers, and asks you to define functions

```
inc   : Bin → Bin
to    : ℕ → Bin
from  : Bin → ℕ
```

Consider the following laws, where `n` ranges over naturals and `b` over bitstrings:

```
from (inc b) ≡ suc (from b)
to (from b) ≡ b
from (to n) ≡ n
```

For each law: if it holds, prove; if not, give a counterexample.

```
-- Your code goes here
```

Standard library

Definitions similar to those in this chapter can be found in the standard library:

```
import Data.Nat.Properties using (+-assoc; +-identityʳ; +-suc; +-comm)
```

Unicode

This chapter uses the following unicode:

```
∀  U+2200  FOR ALL (\forall, \all)
ʳ  U+02B3  MODIFIER LETTER SMALL R (\^r)
′  U+2032  PRIME (\')
″  U+2033  DOUBLE PRIME (\')
‴  U+2034  TRIPLE PRIME (\')
⁗  U+2057  QUADRUPLE PRIME (\')
```

Similar to `\r`, the command `\^r` gives access to a variety of superscript rightward arrows, and also a superscript letter `r`. The command `\'` gives access to a range of primes (`′ ″ ‴ ⁗`).

Chapter 3

Relations: Inductive definition of relations

```
module plfa.part1.Relations where
```

After having defined operations such as addition and multiplication, the next step is to define relations, such as *less than or equal*.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong)
open import Data.Nat using (ℕ; zero; suc; _+_)
open import Data.Nat.Properties using (+-comm; +-identityʳ)
```

Defining relations

The relation *less than or equal* has an infinite number of instances. Here are a few of them:

```
0 ≤ 0     0 ≤ 1     0 ≤ 2     0 ≤ 3     ...
          1 ≤ 1     1 ≤ 2     1 ≤ 3     ...
                    2 ≤ 2     2 ≤ 3     ...
                              3 ≤ 3     ...
                                        ...
```

And yet, we can write a finite definition that encompasses all of these instances in just a few lines. Here is the definition as a pair of inference rules:

```
z≤n --------
    zero ≤ n

    m ≤ n
s≤s -------------
    suc m ≤ suc n
```

And here is the definition in Agda:

```
data _≤_ : ℕ → ℕ → Set where

  z≤n : ∀ {n : ℕ}
      --------
    → zero ≤ n

  s≤s : ∀ {m n : ℕ}
    → m ≤ n
      -------------
    → suc m ≤ suc n
```

Here `z≤n` and `s≤s` (with no spaces) are constructor names, while `zero ≤ n`, and `m ≤ n` and `suc m ≤ suc n` (with spaces) are types. This is our first use of an *indexed* datatype, where the type `m ≤ n` is indexed by two naturals, `m` and `n`. In Agda any line beginning with two or more dashes is a comment, and here we have exploited that convention to write our Agda code in a form that resembles the corresponding inference rules, a trick we will use often from now on.

Both definitions above tell us the same two things:

- *Base case*: for all naturals `n`, the proposition `zero ≤ n` holds.
- *Inductive case*: for all naturals `m` and `n`, if the proposition `m ≤ n` holds, then the proposition `suc m ≤ suc n` holds.

In fact, they each give us a bit more detail:

- *Base case*: for all naturals `n`, the constructor `z≤n` produces evidence that `zero ≤ n` holds.
- *Inductive case*: for all naturals `m` and `n`, the constructor `s≤s` takes evidence that `m ≤ n` holds into evidence that `suc m ≤ suc n` holds.

For example, here in inference rule notation is the proof that `2 ≤ 4`:

```
   z≤n -----
       0 ≤ 2
  s≤s -------
       1 ≤ 3
s≤s ---------
       2 ≤ 4
```

And here is the corresponding Agda proof:

```
_ : 2 ≤ 4
_ = s≤s (s≤s z≤n)
```

Implicit arguments

This is our first use of implicit arguments. In the definition of inequality, the two lines defining the constructors use `∀`, very similar to our use of `∀` in propositions such as:

```
+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
```

However, here the declarations are surrounded by curly braces `{ }` rather than parentheses `( )`. This means that the arguments are *implicit* and need not be written explicitly; instead, they are *inferred* by Agda's typechecker. Thus, we write `+-comm m n` for the proof that `m + n ≡ n + m`,

but `z≤n` for the proof that `zero ≤ n`, leaving `n` implicit. Similarly, if `m≤n` is evidence that `m ≤ n`, we write `s≤s m≤n` for evidence that `suc m ≤ suc n`, leaving both `m` and `n` implicit.

If we wish, it is possible to provide implicit arguments explicitly by writing the arguments inside curly braces. For instance, here is the Agda proof that `2 ≤ 4` repeated, with the implicit arguments made explicit:

```
_ : 2 ≤ 4
_ = s≤s {1} {3} (s≤s {0} {2} (z≤n {2}))
```

One may also identify implicit arguments by name:

```
_ : 2 ≤ 4
_ = s≤s {m = 1} {n = 3} (s≤s {m = 0} {n = 2} (z≤n {n = 2}))
```

In the latter format, you can choose to only supply some implicit arguments:

```
_ : 2 ≤ 4
_ = s≤s {n = 3} (s≤s {n = 2} z≤n)
```

It is not permitted to swap implicit arguments, even when named.

We can ask Agda to use the same inference to try and infer an *explicit* term, by writing `_`. For instance, we can define a variant of the proposition `+-identityʳ` with implicit arguments:

```
+-identityʳ′ : ∀ {m : ℕ} → m + zero ≡ m
+-identityʳ′ = +-identityʳ _
```

We use `_` to ask Agda to infer the value of the *explicit* argument from context. There is only one value which gives us the correct proof, `m`, so Agda happily fills it in. If Agda fails to infer the value, it reports an error.

Precedence

We declare the precedence for comparison as follows:

```
infix 4 _≤_
```

We set the precedence of `_≤_` at level 4, so it binds less tightly than `_+_` at level 6 and hence `1 + 2 ≤ 3` parses as `(1 + 2) ≤ 3`. We write `infix` to indicate that the operator does not associate to either the left or right, as it makes no sense to parse `1 ≤ 2 ≤ 3` as either `(1 ≤ 2) ≤ 3` or `1 ≤ (2 ≤ 3)`.

Decidability

Given two numbers, it is straightforward to compute whether or not the first is less than or equal to the second. We don't give the code for doing so here, but will return to this point in Chapter Decidable.

Inversion

In our definitions, we go from smaller things to larger things. For instance, from `m ≤ n` we can conclude `suc m ≤ suc n`, where `suc m` is bigger than `m` (that is, the former contains the latter), and `suc n` is bigger than `n`. But sometimes we want to go from bigger things to smaller things.

There is only one way to prove that `suc m ≤ suc n`, for any `m` and `n`. This lets us invert our previous rule.

```
inv-s≤s : ∀ {m n : ℕ}
  → suc m ≤ suc n
    --------------
  → m ≤ n
inv-s≤s (s≤s m≤n) = m≤n
```

Here `m≤n` (with no spaces) is a variable name while `m ≤ n` (with spaces) is a type, and the latter is the type of the former. It is a common convention in Agda to derive a variable name by removing spaces from its type.

Not every rule is invertible; indeed, the rule for `z≤n` has no non-implicit hypotheses, so there is nothing to invert. But often inversions of this kind hold.

Another example of inversion is showing that there is only one way a number can be less than or equal to zero.

```
inv-z≤n : ∀ {m : ℕ}
  → m ≤ zero
    --------
  → m ≡ zero
inv-z≤n z≤n = refl
```

Properties of ordering relations

Relations pop up all the time, and mathematicians have agreed on names for some of the most common properties.

- *Reflexive*. For all `n`, the relation `n ≤ n` holds.
- *Transitive*. For all `m`, `n`, and `p`, if `m ≤ n` and `n ≤ p` hold, then `m ≤ p` holds.
- *Anti-symmetric*. For all `m` and `n`, if both `m ≤ n` and `n ≤ m` hold, then `m ≡ n` holds.
- *Total*. For all `m` and `n`, either `m ≤ n` or `n ≤ m` holds.

The relation `_≤_` satisfies all four of these properties.

There are also names for some combinations of these properties.

- *Preorder*. Any relation that is reflexive and transitive.
- *Partial order*. Any preorder that is also anti-symmetric.
- *Total order*. Any partial order that is also total.

If you ever bump into a relation at a party, you now know how to make small talk, by asking it whether it is reflexive, transitive, anti-symmetric, and total. Or instead you might ask whether it is a preorder, partial order, or total order.

Less frivolously, if you ever bump into a relation while reading a technical paper, this gives you a way to orient yourself, by checking whether or not it is a preorder, partial order, or total order. A careful author will often call out these properties—or their lack—for instance by saying that a newly introduced relation is a partial order but not a total order.

Exercise `orderings` (practice)

Give an example of a preorder that is not a partial order.

```
-- Your code goes here
```

Give an example of a partial order that is not a total order.

```
-- Your code goes here
```

Reflexivity

The first property to prove about comparison is that it is reflexive: for any natural `n`, the relation `n ≤ n` holds. We follow the convention in the standard library and make the argument implicit, as that will make it easier to invoke reflexivity:

```
≤-refl : ∀ {n : ℕ}
  -----
  → n ≤ n
≤-refl {zero} = z≤n
≤-refl {suc n} = s≤s ≤-refl
```

The proof is a straightforward induction on the implicit argument `n`. In the base case, `zero ≤ zero` holds by `z≤n`. In the inductive case, the inductive hypothesis `≤-refl {n}` gives us a proof of `n ≤ n`, and applying `s≤s` to that yields a proof of `suc n ≤ suc n`.

It is a good exercise to prove reflexivity interactively in Emacs, using holes and the `C-c C-c`, `C-c C-,`, and `C-c C-r` commands.

Transitivity

The second property to prove about comparison is that it is transitive: for any naturals `m`, `n`, and `p`, if `m ≤ n` and `n ≤ p` hold, then `m ≤ p` holds. Again, `m`, `n`, and `p` are implicit:

```
≤-trans : ∀ {m n p : ℕ}
  → m ≤ n
  → n ≤ p
    -----
  → m ≤ p
≤-trans z≤n _               = z≤n
≤-trans (s≤s m≤n) (s≤s n≤p) = s≤s (≤-trans m≤n n≤p)
```

Here the proof is by induction on the *evidence* that `m ≤ n`. In the base case, the first inequality holds by `z≤n` and must show `zero ≤ p`, which follows immediately by `z≤n`. In this case, the

fact that `n ≤ p` is irrelevant, and we write `_` as the pattern to indicate that the corresponding evidence is unused.

In the inductive case, the first inequality holds by `s≤s m≤n` and the second inequality by `s≤s n≤p`, and so we are given `suc m ≤ suc n` and `suc n ≤ suc p`, and must show `suc m ≤ suc p`. The inductive hypothesis `≤-trans m≤n n≤p` establishes that `m ≤ p`, and our goal follows by applying `s≤s`.

The case `≤-trans (s≤s m≤n) z≤n` cannot arise, since the first inequality implies the middle value is `suc n` while the second inequality implies that it is `zero`. Agda can determine that such a case cannot arise, and does not require (or permit) it to be listed.

Alternatively, we could make the implicit parameters explicit:

```
≤-trans′ : ∀ (m n p : ℕ)
  → m ≤ n
  → n ≤ p
    -----
  → m ≤ p
≤-trans′ zero _ _ z≤n _                                = z≤n
≤-trans′ (suc m) (suc n) (suc p) (s≤s m≤n) (s≤s n≤p) = s≤s (≤-trans′ m n p m≤n n≤p)
```

One might argue that this is clearer or one might argue that the extra length obscures the essence of the proof. We will usually opt for shorter proofs.

The technique of induction on evidence that a property holds (e.g., inducting on evidence that `m ≤ n`)—rather than induction on values of which the property holds (e.g., inducting on `m`)—will turn out to be immensely valuable, and one that we use often.

Again, it is a good exercise to prove transitivity interactively in Emacs, using holes and the `C-c C-c`, `C-c C-,`, and `C-c C-r` commands.

Anti-symmetry

The third property to prove about comparison is that it is antisymmetric: for all naturals `m` and `n`, if both `m ≤ n` and `n ≤ m` hold, then `m ≡ n` holds:

```
≤-antisym : ∀ {m n : ℕ}
  → m ≤ n
  → n ≤ m
    -----
  → m ≡ n
≤-antisym z≤n z≤n             = refl
≤-antisym (s≤s m≤n) (s≤s n≤m) = cong suc (≤-antisym m≤n n≤m)
```

Again, the proof is by induction over the evidence that `m ≤ n` and `n ≤ m` hold.

In the base case, both inequalities hold by `z≤n`, and so we are given `zero ≤ zero` and `zero ≤ zero` and must show `zero ≡ zero`, which follows by reflexivity. (Reflexivity of equality, that is, not reflexivity of inequality.)

In the inductive case, the first inequality holds by `s≤s m≤n` and the second inequality holds by `s≤s n≤m`, and so we are given `suc m ≤ suc n` and `suc n ≤ suc m` and must show `suc m ≡ suc n`. The inductive hypothesis `≤-antisym m≤n n≤m` establishes that `m ≡ n`, and our goal follows by congruence.

Exercise `≤-antisym-cases` (practice)

The above proof omits cases where one argument is `z≤n` and one argument is `s≤s`. Why is it ok to omit them?

```
-- Your code goes here
```

Total

The fourth property to prove about comparison is that it is total: for any naturals `m` and `n` either `m ≤ n` or `n ≤ m`, or both if `m` and `n` are equal.

We specify what it means for inequality to be total:

```
data Total (m n : ℕ) : Set where

  forward :
    m ≤ n
    ---------
    → Total m n

  flipped :
    n ≤ m
    ---------
    → Total m n
```

Evidence that `Total m n` holds is either of the form `forward m≤n` or `flipped n≤m`, where `m≤n` and `n≤m` are evidence of `m ≤ n` and `n ≤ m` respectively.

(For those familiar with logic, the above definition could also be written as a disjunction. Disjunctions will be introduced in Chapter Connectives.)

This is our first use of a datatype with *parameters*, in this case `m` and `n`. It is equivalent to the following indexed datatype:

```
data Total′ : ℕ → ℕ → Set where

  forward′ : ∀ {m n : ℕ}
    → m ≤ n
      ----------
    → Total′ m n

  flipped′ : ∀ {m n : ℕ}
    → n ≤ m
      ----------
    → Total′ m n
```

Each parameter of the type translates as an implicit parameter of each constructor. Unlike an indexed datatype, where the indexes can vary (as in `zero ≤ n` and `suc m ≤ suc n`), in a parameterised datatype the parameters must always be the same (as in `Total m n`). Parameterised declarations are shorter, easier to read, and occasionally aid Agda's termination checker, so we will use them in preference to indexed types when possible.

With that preliminary out of the way, we specify and prove totality:

```
≤-total : ∀ (m n : ℕ) → Total m n
≤-total zero n       = forward z≤n
≤-total (suc m) zero = flipped z≤n
≤-total (suc m) (suc n) with ≤-total m n
... | forward m≤n    = forward (s≤s m≤n)
... | flipped n≤m    = flipped (s≤s n≤m)
```

In this case the proof is by induction over both the first and second arguments. We perform a case analysis:

- *First base case*: If the first argument is `zero` and the second argument is `n` then the forward case holds, with `z≤n` as evidence that `zero ≤ n`.
- *Second base case*: If the first argument is `suc m` and the second argument is `zero` then the flipped case holds, with `z≤n` as evidence that `zero ≤ suc m`.
- *Inductive case*: If the first argument is `suc m` and the second argument is `suc n`, then the inductive hypothesis `≤-total m n` establishes one of the following:
  - The forward case of the inductive hypothesis holds with `m≤n` as evidence that `m ≤ n`, from which it follows that the forward case of the proposition holds with `s≤s m≤n` as evidence that `suc m ≤ suc n`.
  - The flipped case of the inductive hypothesis holds with `n≤m` as evidence that `n ≤ m`, from which it follows that the flipped case of the proposition holds with `s≤s n≤m` as evidence that `suc n ≤ suc m`.

This is our first use of the `with` clause in Agda. The keyword `with` is followed by an expression and one or more subsequent lines. Each line begins with an ellipsis (`...`) and a vertical bar (`|`), followed by a pattern to be matched against the expression and the right-hand side of the equation.

Every use of `with` is equivalent to defining a helper function. For example, the definition above is equivalent to the following:

```
≤-total′ : ∀ (m n : ℕ) → Total m n
≤-total′ zero n          = forward z≤n
≤-total′ (suc m) zero    = flipped z≤n
≤-total′ (suc m) (suc n) = helper (≤-total′ m n)
  where
  helper : Total m n → Total (suc m) (suc n)
  helper (forward m≤n) = forward (s≤s m≤n)
  helper (flipped n≤m) = flipped (s≤s n≤m)
```

This is also our first use of a `where` clause in Agda. The keyword `where` is followed by one or more definitions, which must be indented. Any variables bound on the left-hand side of the preceding equation (in this case, `m` and `n`) are in scope within the nested definition, and any identifiers bound in the nested definition (in this case, `helper`) are in scope in the right-hand side of the preceding equation.

If both arguments are equal, then both cases hold and we could return evidence of either. In the code above we return the forward case, but there is a variant that returns the flipped case:

```
≤-total″ : ∀ (m n : ℕ) → Total m n
≤-total″ m    zero    = flipped z≤n
≤-total″ zero (suc n) = forward z≤n
≤-total″ (suc m) (suc n) with ≤-total″ m n
... | forward m≤n     = forward (s≤s m≤n)
```

```
... | flipped n≤m     = flipped (s≤s n≤m)
```

It differs from the original code in that it pattern matches on the second argument before the first argument.

Monotonicity

If one bumps into both an operator and an ordering at a party, one may ask if the operator is *monotonic* with regard to the ordering. For example, addition is monotonic with regard to inequality, meaning:

```
∀ {m n p q : ℕ} → m ≤ n → p ≤ q → m + p ≤ n + q
```

The proof is straightforward using the techniques we have learned, and is best broken into three parts. First, we deal with the special case of showing addition is monotonic on the right:

```
+-monoʳ-≤ : ∀ (n p q : ℕ)
  → p ≤ q
    -------------
  → n + p ≤ n + q
+-monoʳ-≤ zero p q p≤q     = p≤q
+-monoʳ-≤ (suc n) p q p≤q = s≤s (+-monoʳ-≤ n p q p≤q)
```

The proof is by induction on the first argument.

- *Base case*: The first argument is `zero` in which case `zero + p ≤ zero + q` simplifies to `p ≤ q`, the evidence for which is given by the argument `p≤q`.
- *Inductive case*: The first argument is `suc n`, in which case `suc n + p ≤ suc n + q` simplifies to `suc (n + p) ≤ suc (n + q)`. The inductive hypothesis `+-monoʳ-≤ n p q p≤q` establishes that `n + p ≤ n + q`, and our goal follows by applying `s≤s`.

Second, we deal with the special case of showing addition is monotonic on the left. This follows from the previous result and the commutativity of addition:

```
+-monoˡ-≤ : ∀ (m n p : ℕ)
  → m ≤ n
    -------------
  → m + p ≤ n + p
+-monoˡ-≤ m n p m≤n rewrite +-comm m p | +-comm n p = +-monoʳ-≤ p m n m≤n
```

Rewriting by `+-comm m p` and `+-comm n p` converts `m + p ≤ n + p` into `p + m ≤ p + n`, which is proved by invoking `+-monoʳ-≤ p m n m≤n`.

Third, we combine the two previous results:

```
+-mono-≤ : ∀ (m n p q : ℕ)
  → m ≤ n
  → p ≤ q
    -------------
  → m + p ≤ n + q
+-mono-≤ m n p q m≤n p≤q = ≤-trans (+-monoˡ-≤ m n p m≤n) (+-monoʳ-≤ n p q p≤q)
```

Invoking `+-monoˡ-≤ m n p m≤n` proves `m + p ≤ n + p` and invoking `+-monoʳ-≤ n p q p≤q` proves `n + p ≤ n + q`, and combining these with transitivity proves `m + p ≤ n + q`, as was to be shown.

Exercise `*-mono-≤` (stretch)

Show that multiplication is monotonic with regard to inequality.

```
-- Your code goes here
```

Strict inequality

We can define strict inequality similarly to inequality:

```
infix 4 _<_

data _<_ : ℕ → ℕ → Set where

  z<s : ∀ {n : ℕ}
      ------------
    → zero < suc n

  s<s : ∀ {m n : ℕ}
    → m < n
      -------------
    → suc m < suc n
```

The key difference is that zero is less than the successor of an arbitrary number, but is not less than zero.

Clearly, strict inequality is not reflexive. However it is *irreflexive* in that `n < n` never holds for any value of `n`. Like inequality, strict inequality is transitive. Strict inequality is not total, but satisfies the closely related property of *trichotomy*: for any `m` and `n`, exactly one of `m < n`, `m ≡ n`, or `m > n` holds (where we define `m > n` to hold exactly when `n < m`). It is also monotonic with regards to addition and multiplication.

Most of the above are considered in exercises below. Irreflexivity requires negation, as does the fact that the three cases in trichotomy are mutually exclusive, so those points are deferred to Chapter Negation.

It is straightforward to show that `suc m ≤ n` implies `m < n`, and conversely. One can then give an alternative derivation of the properties of strict inequality, such as transitivity, by exploiting the corresponding properties of inequality.

Exercise `<-trans` (recommended)

Show that strict inequality is transitive.

```
-- Your code goes here
```

Exercise `trichotomy` (practice)

Show that strict inequality satisfies a weak version of trichotomy, in the sense that for any `m` and `n` that one of the following holds: * `m < n`, * `m ≡ n`, or * `m > n`.

Define `m > n` to be the same as `n < m`. You will need a suitable data declaration, similar to that used for totality. (We will show that the three cases are exclusive after we introduce negation.)

```
-- Your code goes here
```

Exercise `+-mono-<` (practice)

Show that addition is monotonic with respect to strict inequality. As with inequality, some additional definitions may be required.

```
-- Your code goes here
```

Exercise `≤-iff-<` (recommended)

Show that `suc m ≤ n` implies `m < n`, and conversely.

```
-- Your code goes here
```

Exercise `<-trans-revisited` (practice)

Give an alternative proof that strict inequality is transitive, using the relation between strict inequality and inequality and the fact that inequality is transitive.

```
-- Your code goes here
```

Even and odd

As a further example, let's specify even and odd numbers. Inequality and strict inequality are *binary relations*, while even and odd are *unary relations*, sometimes called *predicates*:

```
data even : ℕ → Set
data odd : ℕ → Set

data even where

  zero :
     ---------
     even zero

  suc : ∀ {n : ℕ}
    → odd n
      ------------
    → even (suc n)
```

```
data odd where

  suc : ∀ {n : ℕ}
    → even n
      -----------
    → odd (suc n)
```

A number is even if it is zero or the successor of an odd number, and odd if it is the successor of an even number.

This is our first use of a mutually recursive datatype declaration. Since each identifier must be defined before it is used, we first declare the indexed types `even` and `odd` (omitting the `where` keyword and the declarations of the constructors) and then declare the constructors (omitting the signatures `ℕ → Set` which were given earlier).

This is also our first use of *overloaded* constructors, that is, using the same name for constructors of different types. Here `suc` means one of three constructors:

```
suc : ℕ → ℕ

suc : ∀ {n : ℕ}
  → odd n
    ------------
  → even (suc n)

suc : ∀ {n : ℕ}
  → even n
    -----------
  → odd (suc n)
```

Similarly, `zero` refers to one of two constructors. Due to how it does type inference, Agda does not allow overloading of defined names, but does allow overloading of constructors. It is recommended that one restrict overloading to related meanings, as we have done here, but it is not required.

We show that the sum of two even numbers is even:

```
e+e≡e : ∀ {m n : ℕ}
  → even m
  → even n
    ------------
  → even (m + n)

o+e≡o : ∀ {m n : ℕ}
  → odd m
  → even n
    -----------
  → odd (m + n)

e+e≡e zero en     = en
e+e≡e (suc om) en = suc (o+e≡o om en)

o+e≡o (suc em) en = suc (e+e≡e em en)
```

Corresponding to the mutually recursive types, we use two mutually recursive functions, one to show that the sum of two even numbers is even, and the other to show that the sum of an odd and an even number is odd.

This is our first use of mutually recursive functions. Since each identifier must be defined before

it is used, we first give the signatures for both functions and then the equations that define them.

To show that the sum of two even numbers is even, consider the evidence that the first number is even. If it is because it is zero, then the sum is even because the second number is even. If it is because it is the successor of an odd number, then the result is even because it is the successor of the sum of an odd and an even number, which is odd.

To show that the sum of an odd and even number is odd, consider the evidence that the first number is odd. If it is because it is the successor of an even number, then the result is odd because it is the successor of the sum of two even numbers, which is even.

Exercise `o+o≡e` (stretch)

Show that the sum of two odd numbers is even.

```
-- Your code goes here
```

Exercise `Bin-predicates` (stretch)

Recall that Exercise Bin defines a datatype `Bin` of bitstrings representing natural numbers. Representations are not unique due to leading zeros. Hence, eleven may be represented by both of the following:

```
⟨⟩ I O I I
⟨⟩ O O I O I I
```

Define a predicate

```
Can : Bin → Set
```

over all bitstrings that holds if the bitstring is canonical, meaning it has no leading zeros; the first representation of eleven above is canonical, and the second is not. To define it, you will need an auxiliary predicate

```
One : Bin → Set
```

that holds only if the bistring has a leading one. A bitstring is canonical if it has a leading one (representing a positive number) or if it consists of a single zero (representing zero).

Show that increment preserves canonical bitstrings:

```
Can b
------------
Can (inc b)
```

Show that converting a natural to a bitstring always yields a canonical bitstring:

```
----------
Can (to n)
```

Show that converting a canonical bitstring to a natural and back is the identity:

```
Can b
---------------
to (from b) ≡ b
```

(Hint: For each of these, you may first need to prove related properties of `One`. Also, you may need to prove that if `One b` then `1` is less or equal to the result of `from b`.)

```
-- Your code goes here
```

Standard library

Definitions similar to those in this chapter can be found in the standard library:

```
import Data.Nat using (_≤_; z≤n; s≤s)
import Data.Nat.Properties using (≤-refl; ≤-trans; ≤-antisym; ≤-total;
                                  +-monoʳ-≤; +-monoˡ-≤; +-mono-≤)
```

In the standard library, `≤-total` is formalised in terms of disjunction (which we define in Chapter Connectives), and `+-monoʳ-≤`, `+-monoˡ-≤`, `+-mono-≤` are proved differently than here, and more arguments are implicit.

Unicode

This chapter uses the following unicode:

```
≤  U+2264  LESS-THAN OR EQUAL TO (\<=, \le)
≥  U+2265  GREATER-THAN OR EQUAL TO (\>=, \ge)
ˡ  U+02E1  MODIFIER LETTER SMALL L (\^l)
ʳ  U+02B3  MODIFIER LETTER SMALL R (\^r)
```

The commands `\^l` and `\^r` give access to a variety of superscript leftward and rightward arrows in addition to superscript letters `l` and `r`.

Chapter 4

Equality: Equality and equational reasoning

```
module plfa.part1.Equality where
```

Much of our reasoning has involved equality. Given two terms `M` and `N`, both of type `A`, we write `M ≡ N` to assert that `M` and `N` are interchangeable. So far we have treated equality as a primitive, here we show how to define it as an inductive datatype.

Imports

This chapter has no imports. Every chapter in this book, and nearly every module in the Agda standard library, imports equality. Since we define equality here, any import would create a conflict.

Equality

We declare equality as follows:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

In other words, for any type `A` and for any `x` of type `A`, the constructor `refl` provides evidence that `x ≡ x`. Hence, every value is equal to itself, and we have no other way of showing values equal. The definition features an asymmetry, in that the first argument to `_≡_` is given by the parameter `x : A`, while the second is given by an index in `A → Set`. This follows our policy of using parameters wherever possible. The first argument to `_≡_` can be a parameter because it doesn't vary, while the second must be an index, so it can be required to be equal to the first.

We declare the precedence of equality as follows:

```
infix 4 _≡_
```

We set the precedence of `_≡_` at level 4, the same as `_≤_`, which means it binds less tightly than any arithmetic operator. It associates neither to left nor right; writing `x ≡ y ≡ z` is illegal.

Equality is an equivalence relation

An equivalence relation is one which is reflexive, symmetric, and transitive. Reflexivity is built-in to the definition of equality, via the constructor `refl`. It is straightforward to show symmetry:

```
sym : ∀ {A : Set} {x y : A}
  → x ≡ y
    -----
  → y ≡ x
sym refl = refl
```

How does this proof work? The argument to `sym` has type `x ≡ y`, but on the left-hand side of the equation the argument has been instantiated to the pattern `refl`, which requires that `x` and `y` are the same. Hence, for the right-hand side of the equation we need a term of type `x ≡ x`, and `refl` will do.

It is instructive to develop `sym` interactively. To start, we supply a variable for the argument on the left, and a hole for the body on the right:

```
sym : ∀ {A : Set} {x y : A}
  → x ≡ y
    -----
  → y ≡ x
sym e = {! !}
```

If we go into the hole and type `C-c C-,` then Agda reports:

```
Goal: .y ≡ .x
————————————————————————————————————————————————————————————
e  : .x ≡ .y
.y : .A
.x : .A
.A : Set
```

If in the hole we type `C-c C-c e` then Agda will instantiate `e` to all possible constructors, with one equation for each. There is only one possible constructor:

```
sym : ∀ {A : Set} {x y : A}
  → x ≡ y
    -----
  → y ≡ x
sym refl = {! !}
```

If we go into the hole again and type `C-c C-,` then Agda now reports:

```
Goal: .x ≡ .x
————————————————————————————————————————————————————————————
.x : .A
.A : Set
```

This is the key step—Agda has worked out that `x` and `y` must be the same to match the pattern `refl`!

Finally, if we go back into the hole and type `C-c C-r` it will instantiate the hole with the one constructor that yields a value of the expected type:

```
sym : ∀ {A : Set} {x y : A}
  → x ≡ y
    -----
  → y ≡ x
sym refl = refl
```

This completes the definition as given above.

Transitivity is equally straightforward:

```
trans : ∀ {A : Set} {x y z : A}
  → x ≡ y
  → y ≡ z
    -----
  → x ≡ z
trans refl refl = refl
```

Again, a useful exercise is to carry out an interactive development, checking how Agda's knowledge changes as each of the two arguments is instantiated.

Congruence and substitution

Equality satisfies *congruence*. If two terms are equal, they remain so after the same function is applied to both:

```
cong : ∀ {A B : Set} (f : A → B) {x y : A}
  → x ≡ y
    ---------
  → f x ≡ f y
cong f refl = refl
```

Congruence of functions with two arguments is similar:

```
cong₂ : ∀ {A B C : Set} (f : A → B → C) {u x : A} {v y : B}
  → u ≡ x
  → v ≡ y
    -------------
  → f u v ≡ f x y
cong₂ f refl refl = refl
```

Equality is also a congruence in the function position of an application. If two functions are equal, then applying them to the same term yields equal terms:

```
cong-app : ∀ {A B : Set} {f g : A → B}
  → f ≡ g
    ----------------------
  → ∀ (x : A) → f x ≡ g x
cong-app refl x = refl
```

Equality also satisfies *substitution*. If two values are equal and a predicate holds of the first then it also holds of the second:

```
subst : ∀ {A : Set} {x y : A} (P : A → Set)
  → x ≡ y
    ---------
```

```
  → P x → P y
subst P refl px = px
```

Chains of equations

Here we show how to support reasoning with chains of equations, as used throughout the book. We package the declarations into a module, named `≡-Reasoning`, to match the format used in Agda's standard library:

```
module ≡-Reasoning {A : Set} where

  infix 1 begin_
  infixr 2 _≡⟨⟩_ _≡⟨_⟩_
  infix 3 _∎

  begin_ : ∀ {x y : A}
    → x ≡ y
      -----
    → x ≡ y
  begin x≡y = x≡y

  _≡⟨⟩_ : ∀ (x : A) {y : A}
    → x ≡ y
      -----
    → x ≡ y
  x ≡⟨⟩ x≡y = x≡y

  _≡⟨_⟩_ : ∀ (x : A) {y z : A}
    → x ≡ y
    → y ≡ z
      -----
    → x ≡ z
  x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z

  _∎ : ∀ (x : A)
      -----
    → x ≡ x
  x ∎ = refl

open ≡-Reasoning
```

This is our first use of a nested module. It consists of the keyword `module` followed by the module name and any parameters, explicit or implicit, the keyword `where`, and the contents of the module indented. Modules may contain any sort of declaration, including other nested modules. Nested modules are similar to the top-level modules that constitute each chapter of this book, save that the body of a top-level module need not be indented. Opening the module makes all of the definitions available in the current environment.

As an example, let's look at a proof of transitivity as a chain of equations:

```
trans′ : ∀ {A : Set} {x y z : A}
  → x ≡ y
  → y ≡ z
    -----
  → x ≡ z
```

```
trans′ {A} {x} {y} {z} x≡y y≡z =
  begin
    x
  ≡⟨ x≡y ⟩
    y
  ≡⟨ y≡z ⟩
    z
  ∎
```

According to the fixity declarations, the body parses as follows:

```
begin (x ≡⟨ x≡y ⟩ (y ≡⟨ y≡z ⟩ (z ∎)))
```

The application of `begin` is purely cosmetic, as it simply returns its argument. That argument consists of `_≡⟨_⟩_` applied to `x`, `x≡y`, and `y ≡⟨ y≡z ⟩ (z ∎)`. The first argument is a term, `x`, while the second and third arguments are both proofs of equations, in particular proofs of `x ≡ y` and `y ≡ z` respectively, which are combined by `trans` in the body of `_≡⟨_⟩_` to yield a proof of `x ≡ z`. The proof of `y ≡ z` consists of `_≡⟨_⟩_` applied to `y`, `y≡z`, and `z ∎`. The first argument is a term, `y`, while the second and third arguments are both proofs of equations, in particular proofs of `y ≡ z` and `z ≡ z` respectively, which are combined by `trans` in the body of `_≡⟨_⟩_` to yield a proof of `y ≡ z`. Finally, the proof of `z ≡ z` consists of `_∎` applied to the term `z`, which yields `refl`. After simplification, the body is equivalent to the term:

```
trans x≡y (trans y≡z refl)
```

We could replace any use of a chain of equations by a chain of applications of `trans`; the result would be more compact but harder to read. The trick behind `∎` means that a chain of equalities simplifies to a chain of applications of `trans` that ends in `trans e refl`, where `e` is a term that proves some equality, even though `e` alone would do.

Exercise `trans` and `≡-Reasoning` (practice)

Sadly, we cannot use the definition of trans' using ≡-Reasoning as the definition for trans. Can you see why? (Hint: look at the definition of `_≡⟨_⟩_`)

```
-- Your code goes here
```

Chains of equations, another example

As a second example of chains of equations, we repeat the proof that addition is commutative. We first repeat the definitions of naturals and addition. We cannot import them because (as noted at the beginning of this chapter) it would cause a conflict:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero + n    = n
```

```
(suc m) + n = suc (m + n)
```

To save space we postulate (rather than prove in full) two lemmas:

```
postulate
  +-identity : ∀ (m : ℕ) → m + zero ≡ m
  +-suc : ∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
```

This is our first use of a *postulate*. A postulate specifies a signature for an identifier but no definition. Here we postulate something proved earlier to save space. Postulates must be used with caution. If we postulate something false then we could use Agda to prove anything whatsoever.

We then repeat the proof of commutativity:

```
+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
+-comm m zero =
  begin
    m + zero
  ≡⟨ +-identity m ⟩
    m
  ≡⟨⟩
    zero + m
  ∎
+-comm m (suc n) =
  begin
    m + suc n
  ≡⟨ +-suc m n ⟩
    suc (m + n)
  ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
  ≡⟨⟩
    suc n + m
  ∎
```

The reasoning here is similar to that in the preceding section. We use `_≡⟨⟩_` when no justification is required. One can think of `_≡⟨⟩_` as equivalent to `_≡⟨ refl ⟩_`.

Agda always treats a term as equivalent to its simplified term. The reason that one can write

```
  suc (n + m)
≡⟨⟩
  suc n + m
```

is because Agda treats both terms as the same. This also means that one could instead interchange the lines and write

```
  suc n + m
≡⟨⟩
  suc (n + m)
```

and Agda would not object. Agda only checks that the terms separated by `≡⟨⟩` have the same simplified form; it's up to us to write them in an order that will make sense to the reader.

Exercise `≤-Reasoning` (stretch)

The proof of monotonicity from Chapter Relations can be written in a more readable form by using an analogue of our notation for `≡-Reasoning`. Define `≤-Reasoning` analogously, and use it to write out an alternative proof that addition is monotonic with regard to inequality. Rewrite all of `+-monoˡ-≤`, `+-monoʳ-≤`, and `+-mono-≤`.

```
-- Your code goes here
```

Rewriting

Consider a property of natural numbers, such as being even. We repeat the earlier definition:

```
data even : ℕ → Set
data odd : ℕ → Set

data even where

  even-zero : even zero

  even-suc : ∀ {n : ℕ}
    → odd n
      ------------
    → even (suc n)

data odd where
  odd-suc : ∀ {n : ℕ}
    → even n
      -----------
    → odd (suc n)
```

In the previous section, we proved addition is commutative. Given evidence that `even (m + n)` holds, we ought also to be able to take that as evidence that `even (n + m)` holds.

Agda includes special notation to support just this kind of reasoning, the `rewrite` notation we encountered earlier. To enable this notation, we use pragmas to tell Agda which type corresponds to equality:

```
{-# BUILTIN EQUALITY _≡_ #-}
```

We can then prove the desired property as follows:

```
even-comm : ∀ (m n : ℕ)
  → even (m + n)
    ------------
  → even (n + m)
even-comm m n ev rewrite +-comm n m = ev
```

Here `ev` ranges over evidence that `even (m + n)` holds, and we show that it also provides evidence that `even (n + m)` holds. In general, the keyword `rewrite` is followed by evidence of an equality, and that equality is used to rewrite the type of the goal and of any variable in scope.

It is instructive to develop `even-comm` interactively. To start, we supply variables for the arguments on the left, and a hole for the body on the right:

```
even-comm : ∀ (m n : ℕ)
  → even (m + n)
    ------------
  → even (n + m)
even-comm m n ev = {! !}
```

If we go into the hole and type `C-c C-,` then Agda reports:

```
Goal: even (n + m)
————————————————————————————————————————————————————————————
ev : even (m + n)
n  : ℕ
m  : ℕ
```

Now we add the rewrite:

```
even-comm : ∀ (m n : ℕ)
  → even (m + n)
    ------------
  → even (n + m)
even-comm m n ev rewrite +-comm n m = {! !}
```

If we go into the hole again and type `C-c C-,` then Agda now reports:

```
Goal: even (m + n)
————————————————————————————————————————————————————————————
ev : even (m + n)
n  : ℕ
m  : ℕ
```

The arguments have been swapped in the goal. Now it is trivial to see that `ev` satisfies the goal, and typing `C-c C-a` in the hole causes it to be filled with `ev`. The command `C-c C-a` performs an automated search, including checking whether a variable in scope has the same type as the goal.

Multiple rewrites

One may perform multiple rewrites, each separated by a vertical bar. For instance, here is a second proof that addition is commutative, relying on rewrites rather than chains of equalities:

```
+-comm′ : ∀ (m n : ℕ) → m + n ≡ n + m
+-comm′ zero n    rewrite +-identity n          = refl
+-comm′ (suc m) n rewrite +-suc n m | +-comm′ m n = refl
```

This is far more compact. Among other things, whereas the previous proof required `cong suc (+-comm m n)` as the justification to invoke the inductive hypothesis, here it is sufficient to rewrite with `+-comm m n`, as rewriting automatically takes congruence into account. Although proofs with rewriting are shorter, proofs as chains of equalities are easier to follow, and we will stick with the latter when feasible.

Rewriting expanded

The `rewrite` notation is in fact shorthand for an appropriate use of `with` abstraction:

```
even-comm′ : ∀ (m n : ℕ)
  → even (m + n)
    ------------
  → even (n + m)
even-comm′ m n ev with m + n | +-comm m n
... | .(n + m) | refl = ev
```

In general, one can follow `with` by any number of expressions, separated by bars, where each following equation has the same number of patterns. We often write expressions and the corresponding patterns so they line up in columns, as above. Here the first column asserts that `m + n` and `n + m` are identical, and the second column justifies that assertion with evidence of the appropriate equality. Note also the use of the *dot pattern*, `.(n + m)`. A dot pattern consists of a dot followed by an expression, and is used when other information forces the value matched to be equal to the value of the expression in the dot pattern. In this case, the identification of `m + n` and `n + m` is justified by the subsequent matching of `+-comm m n` against `refl`. One might think that the first clause is redundant as the information is inherent in the second clause, but in fact Agda is rather picky on this point: omitting the first clause or reversing the order of the clauses will cause Agda to report an error. (Try it and see!)

In this case, we can avoid rewrite by simply applying the substitution function defined earlier:

```
even-comm″ : ∀ (m n : ℕ)
  → even (m + n)
    ------------
  → even (n + m)
even-comm″ m n = subst even (+-comm m n)
```

Nonetheless, rewrite is a vital part of the Agda toolkit. We will use it sparingly, but it is occasionally essential.

Leibniz equality

The form of asserting equality that we have used is due to Martin-Löf, and was published in 1975. An older form is due to Leibniz, and was published in 1686. Leibniz asserted the *identity of indiscernibles*: two objects are equal if and only if they satisfy the same properties. This principle sometimes goes by the name Leibniz' Law, and is closely related to Spock's Law, "A difference that makes no difference is no difference". Here we define Leibniz equality, and show that two terms satisfy Leibniz equality if and only if they satisfy Martin-Löf equality.

Leibniz equality is usually formalised to state that `x ≐ y` holds if every property `P` that holds of `x` also holds of `y`. Perhaps surprisingly, this definition is sufficient to also ensure the converse, that every property `P` that holds of `y` also holds of `x`.

Let `x` and `y` be objects of type `A`. We say that `x ≐ y` holds if for every predicate `P` over type `A` we have that `P x` implies `P y`:

```
_≐_ : ∀ {A : Set} (x y : A) → Set₁
_≐_ {A} x y = ∀ (P : A → Set) → P x → P y
```

We cannot write the left-hand side of the equation as `x ≐ y`, and instead we write `_≐_ {A} x y` to provide access to the implicit parameter `A` which appears on the right-hand side.

This is our first use of *levels*. We cannot assign `Set` the type `Set`, since this would lead to contradictions such as Russell's Paradox and Girard's Paradox. Instead, there is a hierarchy of types, where `Set : Set₁`, `Set₁ : Set₂`, and so on. In fact, `Set` itself is just an abbreviation for `Set₀`. Since the equation defining `_≐_` mentions `Set` on the right-hand side, the corresponding signature must use `Set₁`. We say a bit more about levels below.

Leibniz equality is reflexive and transitive, where the first follows by a variant of the identity function and the second by a variant of function composition:

```
refl-≐ : ∀ {A : Set} {x : A}
  → x ≐ x
refl-≐ P Px = Px

trans-≐ : ∀ {A : Set} {x y z : A}
  → x ≐ y
  → y ≐ z
    -----
  → x ≐ z
trans-≐ x≐y y≐z P Px = y≐z P (x≐y P Px)
```

Symmetry is less obvious. We have to show that if `P x` implies `P y` for all predicates `P`, then the implication holds the other way round as well:

```
sym-≐ : ∀ {A : Set} {x y : A}
  → x ≐ y
    -----
  → y ≐ x
sym-≐ {A} {x} {y} x≐y P = Qy
  where
    Q : A → Set
    Q z = P z → P x
    Qx : Q x
    Qx = refl-≐ P
    Qy : Q y
    Qy = x≐y Q Qx
```

Given `x ≐ y`, a specific `P`, we have to construct a proof that `P y` implies `P x`. To do so, we instantiate the equality with a predicate `Q` such that `Q z` holds if `P z` implies `P x`. The property `Q x` is trivial by reflexivity, and hence `Q y` follows from `x ≐ y`. But `Q y` is exactly a proof of what we require, that `P y` implies `P x`.

We now show that Martin-Löf equality implies Leibniz equality, and vice versa. In the forward direction, if we know `x ≡ y` we need for any `P` to take evidence of `P x` to evidence of `P y`, which is easy since equality of `x` and `y` implies that any proof of `P x` is also a proof of `P y`:

```
≡-implies-≐ : ∀ {A : Set} {x y : A}
  → x ≡ y
    -----
  → x ≐ y
≡-implies-≐ x≡y P = subst P x≡y
```

This direction follows from substitution, which we showed earlier.

In the reverse direction, given that for any `P` we can take a proof of `P x` to a proof of `P y` we need to show `x ≡ y`:

```
≐-implies-≡ : ∀ {A : Set} {x y : A}
  → x ≐ y
    -----
  → x ≡ y
≐-implies-≡ {A} {x} {y} x≐y = Qy
  where
    Q : A → Set
    Q z = x ≡ z
    Qx : Q x
    Qx = refl
    Qy : Q y
    Qy = x≐y Q Qx
```

The proof is similar to that for symmetry of Leibniz equality. We take `Q` to be the predicate that holds of `z` if `x ≡ z`. Then `Q x` is trivial by reflexivity of Martin-Löf equality, and hence `Q y` follows from `x ≐ y`. But `Q y` is exactly a proof of what we require, that `x ≡ y`.

(Parts of this section are adapted from *≐≃≡: Leibniz Equality is Isomorphic to Martin-Löf Identity, Parametrically*, by Andreas Abel, Jesper Cockx, Dominique Devries, Andreas Nuyts, and Philip Wadler, draft, 2017.)

Universe polymorphism

As we have seen, not every type belongs to `Set`, but instead every type belongs somewhere in the hierarchy `Set₀`, `Set₁`, `Set₂`, and so on, where `Set` abbreviates `Set₀`, and `Set₀ : Set₁`, `Set₁ : Set₂`, and so on. The definition of equality given above is fine if we want to compare two values of a type that belongs to `Set`, but what if we want to compare two values of a type that belongs to `Set ℓ` for some arbitrary level `ℓ`?

The answer is *universe polymorphism*, where a definition is made with respect to an arbitrary level `ℓ`. To make use of levels, we first import the following:

```
open import Level using (Level; _⊔_) renaming (zero to lzero; suc to lsuc)
```

We rename constructors `zero` and `suc` to `lzero` and `lsuc` to avoid confusion between levels and naturals.

Levels are isomorphic to natural numbers, and have similar constructors:

```
lzero : Level
lsuc  : Level → Level
```

The names `Set₀`, `Set₁`, `Set₂`, and so on, are abbreviations for

```
Set lzero
Set (lsuc lzero)
Set (lsuc (lsuc lzero))
```

and so on. There is also an operator

```
_⊔_ : Level → Level → Level
```

that given two levels returns the larger of the two.

Here is the definition of equality, generalised to an arbitrary level:

```
data _≡′_ {ℓ : Level} {A : Set ℓ} (x : A) : A → Set ℓ where
  refl′ : x ≡′ x
```

Similarly, here is the generalised definition of symmetry:

```
sym′ : ∀ {ℓ : Level} {A : Set ℓ} {x y : A}
  → x ≡′ y
    ------
  → y ≡′ x
sym′ refl′ = refl′
```

For simplicity, we avoid universe polymorphism in the definitions given in the text, but most definitions in the standard library, including those for equality, are generalised to arbitrary levels as above.

Here is the generalised definition of Leibniz equality:

```
_≐′_ : ∀ {ℓ : Level} {A : Set ℓ} (x y : A) → Set (lsuc ℓ)
_≐′_ {ℓ} {A} x y = ∀ (P : A → Set ℓ) → P x → P y
```

Before the signature used `Set₁` as the type of a term that includes `Set`, whereas here the signature uses `Set (lsuc ℓ)` as the type of a term that includes `Set ℓ`.

Most other functions in the standard library are also generalised to arbitrary levels. For instance, here is the definition of composition.

```
_∘_ : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Set ℓ₁} {B : Set ℓ₂} {C : Set ℓ₃}
  → (B → C) → (A → B) → A → C
(g ∘ f) x = g (f x)
```

Further information on levels can be found in the Agda docs.

Standard library

Definitions similar to those in this chapter can be found in the standard library. The Agda standard library defines `_≡⟨_⟩_` as `step-≡`, which reverses the order of the arguments. The standard library also defines a syntax macro, which is automatically imported whenever you import `step-≡`, which recovers the original argument order:

```
-- import Relation.Binary.PropositionalEquality as Eq
-- open Eq using (_≡_; refl; trans; sym; cong; cong-app; subst)
-- open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _∎)
```

Here the imports are shown as comments rather than code to avoid collisions, as mentioned in the introduction.

Unicode

This chapter uses the following unicode:

```
≡  U+2261  IDENTICAL TO (\==, \equiv)
⟨  U+27E8  MATHEMATICAL LEFT ANGLE BRACKET (\<)
⟩  U+27E9  MATHEMATICAL RIGHT ANGLE BRACKET (\>)
∎  U+220E  END OF PROOF (\qed)
≐  U+2250  APPROACHES THE LIMIT (\.=)
ℓ  U+2113  SCRIPT SMALL L (\ell)
⊔  U+2294  SQUARE CUP (\lub)
₀  U+2080  SUBSCRIPT ZERO (\_0)
₁  U+2081  SUBSCRIPT ONE (\_1)
₂  U+2082  SUBSCRIPT TWO (\_2)
```

Chapter 5

Isomorphism: Isomorphism and Embedding

```
module plfa.part1.Isomorphism where
```

This section introduces isomorphism as a way of asserting that two types are equal, and embedding as a way of asserting that one type is smaller than another. We apply isomorphisms in the next chapter to demonstrate that operations on types such as product and sum satisfy properties akin to associativity, commutativity, and distributivity.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; cong-app)
open Eq.≡-Reasoning
open import Data.Nat using (ℕ; zero; suc; _+_)
open import Data.Nat.Properties using (+-comm)
```

Lambda expressions

The chapter begins with a few preliminaries that will be useful here and elsewhere: lambda expressions, function composition, and extensionality.

Lambda expressions provide a compact way to define functions without naming them. A term of the form

```
λ{ P₁ → N₁; ⋯ ; Pₙ → Nₙ }
```

is equivalent to a function `f` defined by the equations

```
f P₁ = N₁
⋯
f Pₙ = Nₙ
```

where the `Pₙ` are patterns (left-hand sides of an equation) and the `Nₙ` are expressions (right-hand side of an equation).

In the case that there is one equation and the pattern is a variable, we may also use the syntax

```
λ x → N
```

or

```
λ (x : A) → N
```

both of which are equivalent to `λ{x → N}`. The latter allows one to specify the domain of the function.

Often using an anonymous lambda expression is more convenient than using a named function: it avoids a lengthy type declaration; and the definition appears exactly where the function is used, so there is no need for the writer to remember to declare it in advance, or for the reader to search for the definition in the code.

Function composition

In what follows, we will make use of function composition:

```
_∘_ : ∀ {A B C : Set} → (B → C) → (A → B) → (A → C)
(g ∘ f) x = g (f x)
```

Thus, `g ∘ f` is the function that first applies `f` and then applies `g`. An equivalent definition, exploiting lambda expressions, is as follows:

```
_∘′_ : ∀ {A B C : Set} → (B → C) → (A → B) → (A → C)
g ∘′ f = λ x → g (f x)
```

Extensionality

Extensionality asserts that the only way to distinguish functions is by applying them; if two functions applied to the same argument always yield the same result, then they are the same function. It is the converse of `cong-app`, as introduced earlier.

Agda does not presume extensionality, but we can postulate that it holds:

```
postulate
  extensionality : ∀ {A B : Set} {f g : A → B}
    → (∀ (x : A) → f x ≡ g x)
      ----------------------
    → f ≡ g
```

Postulating extensionality does not lead to difficulties, as it is known to be consistent with the theory that underlies Agda.

As an example, consider that we need results from two libraries, one where addition is defined, as in Chapter Naturals, and one where it is defined the other way around.

```
_+′_ : ℕ → ℕ → ℕ
m +′ zero = m
m +′ suc n = suc (m +′ n)
```

Applying commutativity, it is easy to show that both operators always return the same result given the same arguments:

```
same-app : ∀ (m n : ℕ) → m +′ n ≡ m + n
same-app m n rewrite +-comm m n = helper m n
  where
  helper : ∀ (m n : ℕ) → m +′ n ≡ n + m
  helper m zero = refl
  helper m (suc n) = cong suc (helper m n)
```

However, it might be convenient to assert that the two operators are actually indistinguishable. This we can do via two applications of extensionality:

```
same : _+′_ ≡ _+_
same = extensionality (λ m → extensionality (λ n → same-app m n))
```

We occasionally need to postulate extensionality in what follows.

More generally, we may wish to postulate extensionality for dependent functions.

```
postulate
  ∀-extensionality : ∀ {A : Set} {B : A → Set} {f g : ∀(x : A) → B x}
    → (∀ (x : A) → f x ≡ g x)
      -----------------------
    → f ≡ g
```

Here the type of `f` and `g` has changed from `A → B` to `∀ (x : A) → B x`, generalising ordinary functions to dependent functions.

Isomorphism

Two sets are isomorphic if they are in one-to-one correspondence. Here is a formal definition of isomorphism:

```
infix 0 _≃_
record _≃_ (A B : Set) : Set where
  field
    to : A → B
    from : B → A
    from∘to : ∀ (x : A) → from (to x) ≡ x
    to∘from : ∀ (y : B) → to (from y) ≡ y
open _≃_
```

Let's unpack the definition. An isomorphism between sets `A` and `B` consists of four things: + A function `to` from `A` to `B`, + A function `from` from `B` back to `A`, + Evidence `from∘to` asserting that `from` is a *left-inverse* for `to`, + Evidence `to∘from` asserting that `from` is a *right-inverse* for `to`.

In particular, the third asserts that `from ∘ to` is the identity, and the fourth that `to ∘ from` is the identity, hence the names. The declaration `open _≃_` makes available the names `to`, `from`,

`from∘to`, and `to∘from`, otherwise we would need to write `_≃_.to` and so on.

The above is our first use of records. A record declaration behaves similar to a single-constructor data declaration (there are minor differences, which we discuss in Connectives):

```
data _≃′_ (A B : Set): Set where
  mk-≃′ : ∀ (to : A → B) →
          ∀ (from : B → A) →
          ∀ (from∘to : (∀ (x : A) → from (to x) ≡ x)) →
          ∀ (to∘from : (∀ (y : B) → to (from y) ≡ y)) →
          A ≃′ B

to′ : ∀ {A B : Set} → (A ≃′ B) → (A → B)
to′ (mk-≃′ f g g∘f f∘g) = f

from′ : ∀ {A B : Set} → (A ≃′ B) → (B → A)
from′ (mk-≃′ f g g∘f f∘g) = g

from∘to′ : ∀ {A B : Set} → (A≃B : A ≃′ B) → (∀ (x : A) → from′ A≃B (to′ A≃B x) ≡ x)
from∘to′ (mk-≃′ f g g∘f f∘g) = g∘f

to∘from′ : ∀ {A B : Set} → (A≃B : A ≃′ B) → (∀ (y : B) → to′ A≃B (from′ A≃B y) ≡ y)
to∘from′ (mk-≃′ f g g∘f f∘g) = f∘g
```

We construct values of the record type with the syntax

```
record
  { to    = f
  ; from  = g
  ; from∘to = g∘f
  ; to∘from = f∘g
  }
```

which corresponds to using the constructor of the corresponding inductive type

```
mk-≃′ f g g∘f f∘g
```

where `f`, `g`, `g∘f`, and `f∘g` are values of suitable types.

Isomorphism is an equivalence

Isomorphism is an equivalence, meaning that it is reflexive, symmetric, and transitive. To show isomorphism is reflexive, we take both `to` and `from` to be the identity function:

```
≃-refl : ∀ {A : Set}
   -----
  → A ≃ A
≃-refl =
  record
    { to   = λ{x → x}
    ; from = λ{y → y}
    ; from∘to = λ{x → refl}
    ; to∘from = λ{y → refl}
    }
```

In the above, `to` and `from` are both bound to identity functions, and `from∘to` and `to∘from` are

both bound to functions that discard their argument and return `refl`. In this case, `refl` alone is an adequate proof since for the left inverse, `from (to x)` simplifies to `x`, and similarly for the right inverse.

To show isomorphism is symmetric, we simply swap the roles of `to` and `from`, and `from∘to` and `to∘from`:

```
≃-sym : ∀ {A B : Set}
  → A ≃ B
    -----
  → B ≃ A
≃-sym A≃B =
  record
    { to   = from A≃B
    ; from = to A≃B
    ; from∘to = to∘from A≃B
    ; to∘from = from∘to A≃B
    }
```

To show isomorphism is transitive, we compose the `to` and `from` functions, and use equational reasoning to combine the inverses:

```
≃-trans : ∀ {A B C : Set}
  → A ≃ B
  → B ≃ C
    -----
  → A ≃ C
≃-trans A≃B B≃C =
  record
    { to      = to B≃C ∘ to A≃B
    ; from    = from A≃B ∘ from B≃C
    ; from∘to = λ{x →
        begin
          (from A≃B ∘ from B≃C) ((to B≃C ∘ to A≃B) x)
        ≡⟨⟩
          from A≃B (from B≃C (to B≃C (to A≃B x)))
        ≡⟨ cong (from A≃B) (from∘to B≃C (to A≃B x)) ⟩
          from A≃B (to A≃B x)
        ≡⟨ from∘to A≃B x ⟩
          x
        ∎}
    ; to∘from = λ{y →
        begin
          (to B≃C ∘ to A≃B) ((from A≃B ∘ from B≃C) y)
        ≡⟨⟩
          to B≃C (to A≃B (from A≃B (from B≃C y)))
        ≡⟨ cong (to B≃C) (to∘from A≃B (from B≃C y)) ⟩
          to B≃C (from B≃C y)
        ≡⟨ to∘from B≃C y ⟩
          y
        ∎}
    }
```

Equational reasoning for isomorphism

It is straightforward to support a variant of equational reasoning for isomorphism. We essentially copy the previous definition of equality for isomorphism. We omit the form that corresponds to `_≡⟨⟩_`, since trivial isomorphisms arise far less often than trivial equalities:

```
module ≃-Reasoning where

  infix 1 ≃-begin_
  infixr 2 _≃⟨_⟩_
  infix 3 _≃-∎

  ≃-begin_ : ∀ {A B : Set}
    → A ≃ B
      -----
    → A ≃ B
  ≃-begin A≃B = A≃B

  _≃⟨_⟩_ : ∀ (A : Set) {B C : Set}
    → A ≃ B
    → B ≃ C
      -----
    → A ≃ C
  A ≃⟨ A≃B ⟩ B≃C = ≃-trans A≃B B≃C

  _≃-∎ : ∀ (A : Set)
      -----
    → A ≃ A
  A ≃-∎ = ≃-refl

open ≃-Reasoning
```

Embedding

We also need the notion of *embedding*, which is a weakening of isomorphism. While an isomorphism shows that two types are in one-to-one correspondence, an embedding shows that the first type is included in the second; or, equivalently, that there is a many-to-one correspondence between the second type and the first.

Here is the formal definition of embedding:

```
infix 0 _≲_
record _≲_ (A B : Set) : Set where
  field
    to   : A → B
    from : B → A
    from∘to : ∀ (x : A) → from (to x) ≡ x
open _≲_
```

It is the same as an isomorphism, save that it lacks the `to∘from` field. Hence, we know that `from` is left-inverse to `to`, but not that `from` is right-inverse to `to`.

Embedding is reflexive and transitive, but not symmetric. The proofs are cut down versions of the similar proofs for isomorphism:

```
≲-refl : ∀ {A : Set} → A ≲ A
≲-refl =
  record
    { to   = λ{x → x}
    ; from = λ{y → y}
    ; from∘to = λ{x → refl}
    }

≲-trans : ∀ {A B C : Set} → A ≲ B → B ≲ C → A ≲ C
≲-trans A≲B B≲C =
  record
    { to   = λ{x → to B≲C (to A≲B x)}
    ; from = λ{y → from A≲B (from B≲C y)}
    ; from∘to = λ{x →
       begin
         from A≲B (from B≲C (to B≲C (to A≲B x)))
       ≡⟨ cong (from A≲B) (from∘to B≲C (to A≲B x)) ⟩
         from A≲B (to A≲B x)
       ≡⟨ from∘to A≲B x ⟩
         x
       ∎}
     }
```

It is also easy to see that if two types embed in each other, and the embedding functions correspond, then they are isomorphic. This is a weak form of anti-symmetry:

```
≲-antisym : ∀ {A B : Set}
  → (A≲B : A ≲ B)
  → (B≲A : B ≲ A)
  → (to A≲B ≡ from B≲A)
  → (from A≲B ≡ to B≲A)
    -------------------
  → A ≃ B
≲-antisym A≲B B≲A to≡from from≡to =
  record
    { to   = to A≲B
    ; from = from A≲B
    ; from∘to = from∘to A≲B
    ; to∘from = λ{y →
       begin
         to A≲B (from A≲B y)
       ≡⟨ cong (to A≲B) (cong-app from≡to y) ⟩
         to A≲B (to B≲A y)
       ≡⟨ cong-app to≡from (to B≲A y) ⟩
         from B≲A (to B≲A y)
       ≡⟨ from∘to B≲A y ⟩
         y
       ∎}
    }
```

The first three components are copied from the embedding, while the last combines the left inverse of `B ≲ A` with the equivalences of the `to` and `from` components from the two embeddings to obtain the right inverse of the isomorphism.

Equational reasoning for embedding

We can also support tabular reasoning for embedding, analogous to that used for isomorphism:

```
module ≲-Reasoning where

  infix 1 ≲-begin_
  infixr 2 _≲⟨_⟩_
  infix 3 _≲-∎

  ≲-begin_ : ∀ {A B : Set}
    → A ≲ B
      -----
    → A ≲ B
  ≲-begin A≲B = A≲B

  _≲⟨_⟩_ : ∀ (A : Set) {B C : Set}
    → A ≲ B
    → B ≲ C
      -----
    → A ≲ C
  A ≲⟨ A≲B ⟩ B≲C = ≲-trans A≲B B≲C

  _≲-∎ : ∀ (A : Set)
      -----
    → A ≲ A
  A ≲-∎ = ≲-refl

open ≲-Reasoning
```

Exercise `≃-implies-≲` (practice)

Show that every isomorphism implies an embedding.

```
postulate
  ≃-implies-≲ : ∀ {A B : Set}
    → A ≃ B
      -----
    → A ≲ B
```

```
-- Your code goes here
```

Exercise `_⇔_` (practice)

Define equivalence of propositions (also known as "if and only if") as follows:

```
record _⇔_ (A B : Set) : Set where
  field
    to : A → B
    from : B → A
```

Show that equivalence is reflexive, symmetric, and transitive.

```
-- Your code goes here
```

Exercise `Bin-embedding` (stretch)

Recall that Exercises Bin and Bin-laws define a datatype `Bin` of bitstrings representing natural numbers, and asks you to define the following functions and predicates:

```
to : ℕ → Bin
from : Bin → ℕ
```

which satisfy the following property:

```
from (to n) ≡ n
```

Using the above, establish that there is an embedding of `ℕ` into `Bin`.

```
-- Your code goes here
```

Why do `to` and `from` not form an isomorphism?

Standard library

Definitions similar to those in this chapter can be found in the standard library:

```
import Function using (_∘_)
import Function.Inverse using (_↔_)
import Function.LeftInverse using (_↞_)
```

The standard library `_↔_` and `_↞_` correspond to our `_≃_` and `_≲_`, respectively, but those in the standard library are less convenient, since they depend on a nested record structure and are parameterised with regard to an arbitrary notion of equivalence.

Unicode

This chapter uses the following unicode:

```
∘  U+2218  RING OPERATOR (\o, \circ, \comp)
λ  U+03BB  GREEK SMALL LETTER LAMBDA (\lambda, \Gl)
≃  U+2243  ASYMPTOTICALLY EQUAL TO (\~-)
≲  U+2272  LESS-THAN OR EQUIVALENT TO (\<~)
⇔  U+21D4  LEFT RIGHT DOUBLE ARROW (\<=>)
```

Chapter 6

Connectives: Conjunction, disjunction, and implication

```
module plfa.part1.Connectives where
```

This chapter introduces the basic logical connectives, by observing a correspondence between connectives of logic and data types, a principle known as *Propositions as Types*:

- *conjunction* is *product*,
- *disjunction* is *sum*,
- *true* is *unit type*,
- *false* is *empty type*,
- *implication* is *function space*.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open Eq.≡-Reasoning
open import Data.Nat using (ℕ)
open import Function using (_∘_)
open import plfa.part1.Isomorphism using (_≃_; _≲_; extensionality)
open plfa.part1.Isomorphism.≃-Reasoning
```

Conjunction is product

Given two propositions `A` and `B`, the conjunction `A × B` holds if both `A` holds and `B` holds. We formalise this idea by declaring a suitable record type:

```
data _×_ (A B : Set) : Set where

  ⟨_,_⟩ :
    A
    → B
      -----
```

```
  → A × B
```

Evidence that `A × B` holds is of the form `⟨ M , N ⟩`, where `M` provides evidence that `A` holds and `N` provides evidence that `B` holds.

Given evidence that `A × B` holds, we can conclude that both `A` holds and `B` holds:

```
proj₁ : ∀ {A B : Set}
  → A × B
    -----
  → A
proj₁ ⟨ x , y ⟩ = x

proj₂ : ∀ {A B : Set}
  → A × B
    -----
  → B
proj₂ ⟨ x , y ⟩ = y
```

If `L` provides evidence that `A × B` holds, then `proj₁ L` provides evidence that `A` holds, and `proj₂ L` provides evidence that `B` holds.

When `⟨_,_⟩` appears in a term on the right-hand side of an equation we refer to it as a *constructor*, and when it appears in a pattern on the left-hand side of an equation we refer to it as a *destructor*. We may also refer to `proj₁` and `proj₂` as destructors, since they play a similar role.

Other terminology refers to `⟨_,_⟩` as *introducing* a conjunction, and to `proj₁` and `proj₂` as *eliminating* a conjunction; indeed, the former is sometimes given the name `×-I` and the latter two the names `×-E₁` and `×-E₂`. As we read the rules from top to bottom, introduction and elimination do what they say on the tin: the first *introduces* a formula for the connective, which appears in the conclusion but not in the hypotheses; the second *eliminates* a formula for the connective, which appears in a hypothesis but not in the conclusion. An introduction rule describes under what conditions we say the connective holds—how to *define* the connective. An elimination rule describes what we may conclude when the connective holds—how to *use* the connective.[1]

In this case, applying each destructor and reassembling the results with the constructor is the identity over products:

```
η-× : ∀ {A B : Set} (w : A × B) → ⟨ proj₁ w , proj₂ w ⟩ ≡ w
η-× ⟨ x , y ⟩ = refl
```

The pattern matching on the left-hand side is essential, since replacing `w` by `⟨ x , y ⟩` allows both sides of the propositional equality to simplify to the same term.

We set the precedence of conjunction so that it binds less tightly than anything save disjunction:

```
infixr 2 _×_
```

Thus, `m ≤ n × n ≤ p` parses as `(m ≤ n) × (n ≤ p)`.

Alternatively, we can declare conjunction as a record type:

[1] This paragraph was adopted from "Propositions as Types", Philip Wadler, *Communications of the ACM*, December 2015.

```
record _×′_ (A B : Set) : Set where
  constructor ⟨_,_⟩′
  field
    proj₁′ : A
    proj₂′ : B
open _×′_
```

The record construction `record { proj₁′ = M ; proj₂′ = N }` corresponds to the term `⟨ M , N ⟩` where `M` is a term of type `A` and `N` is a term of type `B`. The constructor declaration allows us to write `⟨ M , N ⟩′` in place of the record construction.

The data type `_x_` and the record type `_×′_` behave similarly. One difference is that for data types we have to prove η-equality, but for record types, η-equality holds *by definition*. While proving `η-×′`, we do not have to pattern match on `w` to know that η-equality holds:

```
η-×′ : ∀ {A B : Set} (w : A ×′ B) → ⟨ proj₁′ w , proj₂′ w ⟩′ ≡ w
η-×′ w = refl
```

It can be very convenient to have η-equality *definitionally*, and so the standard library defines `_×_` as a record type. We use the definition from the standard library in later chapters.

Given two types `A` and `B`, we refer to `A × B` as the *product* of `A` and `B`. In set theory, it is also sometimes called the *Cartesian product*, and in computing it corresponds to a *record* type. Among other reasons for calling it the product, note that if type `A` has `m` distinct members, and type `B` has `n` distinct members, then the type `A × B` has `m * n` distinct members. For instance, consider a type `Bool` with two members, and a type `Tri` with three members:

```
data Bool : Set where
  true : Bool
  false : Bool

data Tri : Set where
  aa : Tri
  bb : Tri
  cc : Tri
```

Then the type `Bool × Tri` has six members:

```
⟨ true  , aa ⟩    ⟨ true  , bb ⟩    ⟨ true ,  cc ⟩
⟨ false , aa ⟩    ⟨ false , bb ⟩    ⟨ false , cc ⟩
```

For example, the following function enumerates all possible arguments of type `Bool × Tri`:

```
×-count : Bool × Tri → ℕ
×-count ⟨ true , aa ⟩  = 1
×-count ⟨ true , bb ⟩  = 2
×-count ⟨ true , cc ⟩  = 3
×-count ⟨ false , aa ⟩ = 4
×-count ⟨ false , bb ⟩ = 5
×-count ⟨ false , cc ⟩ = 6
```

Product on types also shares a property with product on numbers in that there is a sense in which it is commutative and associative. In particular, product is commutative and associative *up to isomorphism*.

For commutativity, the `to` function swaps a pair, taking `⟨ x , y ⟩` to `⟨ y , x ⟩`, and the `from`

function does the same (up to renaming). Instantiating the patterns correctly in `from∘to` and `to∘from` is essential. Replacing the definition of `from∘to` by `λ w → refl` will not work; and similarly for `to∘from`:

```
×-comm : ∀ {A B : Set} → A × B ≃ B × A
×-comm =
  record
    { to      = λ{ ⟨ x , y ⟩ → ⟨ y , x ⟩ }
    ; from    = λ{ ⟨ y , x ⟩ → ⟨ x , y ⟩ }
    ; from∘to = λ{ ⟨ x , y ⟩ → refl }
    ; to∘from = λ{ ⟨ y , x ⟩ → refl }
    }
```

Being *commutative* is different from being *commutative up to isomorphism*. Compare the two statements:

```
m * n ≡ n * m
A × B ≃ B × A
```

In the first case, we might have that `m` is `2` and `n` is `3`, and both `m * n` and `n * m` are equal to `6`. In the second case, we might have that `A` is `Bool` and `B` is `Tri`, and `Bool × Tri` is *not* the same as `Tri × Bool`. But there is an isomorphism between the two types. For instance, `⟨ true , aa ⟩`, which is a member of the former, corresponds to `⟨ aa , true ⟩`, which is a member of the latter.

For associativity, the `to` function reassociates two uses of pairing, taking `⟨ ⟨ x , y ⟩ , z ⟩` to `⟨ x , ⟨ y , z ⟩ ⟩`, and the `from` function does the inverse. Again, the evidence of left and right inverse requires matching against a suitable pattern to enable simplification:

```
×-assoc : ∀ {A B C : Set} → (A × B) × C ≃ A × (B × C)
×-assoc =
  record
    { to   = λ{ ⟨ ⟨ x , y ⟩ , z ⟩ → ⟨ x , ⟨ y , z ⟩ ⟩ }
    ; from = λ{ ⟨ x , ⟨ y , z ⟩ ⟩ → ⟨ ⟨ x , y ⟩ , z ⟩ }
    ; from∘to = λ{ ⟨ ⟨ x , y ⟩ , z ⟩ → refl }
    ; to∘from = λ{ ⟨ x , ⟨ y , z ⟩ ⟩ → refl }
    }
```

Being *associative* is not the same as being *associative up to isomorphism*. Compare the two statements:

```
(m * n) * p ≡ m * (n * p)
(A × B) × C ≃ A × (B × C)
```

For example, the type `(ℕ × Bool) × Tri` is *not* the same as `ℕ × (Bool × Tri)`. But there is an isomorphism between the two types. For instance `⟨ ⟨ 1 , true ⟩ , aa ⟩`, which is a member of the former, corresponds to `⟨ 1 , ⟨ true , aa ⟩ ⟩`, which is a member of the latter.

Exercise `⇔≃×` (recommended)

Show that `A ⇔ B` as defined earlier is isomorphic to `(A → B) × (B → A)`.

```
-- Your code goes here
```

Truth is unit

Truth `⊤` always holds. We formalise this idea by declaring a suitable record type:

```
data ⊤ : Set where

  tt :
    --
    ⊤
```

Evidence that `⊤` holds is of the form `tt`.

There is an introduction rule, but no elimination rule. Given evidence that `⊤` holds, there is nothing more of interest we can conclude. Since truth always holds, knowing that it holds tells us nothing new.

The nullary case of `η-×` is `η-⊤`, which asserts that any value of type `⊤` must be equal to `tt`:

```
η-⊤ : ∀ (w : ⊤) → tt ≡ w
η-⊤ tt = refl
```

The pattern matching on the left-hand side is essential. Replacing `w` by `tt` allows both sides of the propositional equality to simplify to the same term.

Alternatively, we can declare truth as an empty record:

```
record ⊤′ : Set where
  constructor tt′
```

The record construction `record {}` corresponds to the term `tt`. The constructor declaration allows us to write `tt′`.

As with the product, the data type `⊤` and the record type `⊤′` behave similarly, but η-equality holds *by definition* for the record type. While proving `η-⊤′`, we do not have to pattern match on `w`—Agda *knows* it is equal to `tt′`:

```
η-⊤′ : ∀ (w : ⊤′) → tt′ ≡ w
η-⊤′ w = refl
```

Agda knows that *any* value of type `⊤′` must be `tt′`, so any time we need a value of type `⊤′`, we can tell Agda to figure it out:

```
truth′ : ⊤′
truth′ = _
```

We refer to `⊤` as the *unit* type. And, indeed, type `⊤` has exactly one member, `tt`. For example, the following function enumerates all possible arguments of type `⊤`:

```
⊤-count : ⊤ → ℕ
⊤-count tt = 1
```

For numbers, one is the identity of multiplication. Correspondingly, unit is the identity of product *up to isomorphism*. For left identity, the `to` function takes `⟨ tt , x ⟩` to `x`, and the `from` function does the inverse. The evidence of left inverse requires matching against a suitable pattern to enable simplification:

```
⊤-identityˡ : ∀ {A : Set} → ⊤ × A ≃ A
⊤-identityˡ =
  record
    { to   = λ{ ⟨ tt , x ⟩ → x }
    ; from = λ{ x → ⟨ tt , x ⟩ }
    ; from∘to = λ{ ⟨ tt , x ⟩ → refl }
    ; to∘from = λ{ x → refl }
    }
```

Having an *identity* is different from having an identity *up to isomorphism*. Compare the two statements:

```
1 * m ≡ m
⊤ × A ≃ A
```

In the first case, we might have that `m` is `2`, and both `1 * m` and `m` are equal to `2`. In the second case, we might have that `A` is `Bool`, and `⊤ × Bool` is *not* the same as `Bool`. But there is an isomorphism between the two types. For instance, `⟨ tt , true ⟩`, which is a member of the former, corresponds to `true`, which is a member of the latter.

Right identity follows from commutativity of product and left identity:

```
⊤-identityʳ : ∀ {A : Set} → (A × ⊤) ≃ A
⊤-identityʳ {A} =
  ≃-begin
    (A × ⊤)
  ≃⟨ ×-comm ⟩
    (⊤ × A)
  ≃⟨ ⊤-identityˡ ⟩
    A
  ≃-∎
```

Here we have used a chain of isomorphisms, analogous to that used for equality.

Disjunction is sum

Given two propositions `A` and `B`, the disjunction `A ⊎ B` holds if either `A` holds or `B` holds. We formalise this idea by declaring a suitable inductive type:

```
data _⊎_ (A B : Set) : Set where

  inj₁ :
    A
    -----
    → A ⊎ B

  inj₂ :
    B
    -----
    → A ⊎ B
```

Evidence that `A ⊎ B` holds is either of the form `inj₁ M`, where `M` provides evidence that `A` holds, or `inj₂ N`, where `N` provides evidence that `B` holds.

Given evidence that `A → C` and `B → C` both hold, then given evidence that `A ⊎ B` holds we can conclude that `C` holds:

```
case-⊎ : ∀ {A B C : Set}
  → (A → C)
  → (B → C)
  → A ⊎ B
    -----------
  → C
case-⊎ f g (inj₁ x) = f x
case-⊎ f g (inj₂ y) = g y
```

Pattern matching against `inj₁` and `inj₂` is typical of how we exploit evidence that a disjunction holds.

When `inj₁` and `inj₂` appear on the right-hand side of an equation we refer to them as *constructors*, and when they appear on the left-hand side we refer to them as *destructors*. We also refer to `case-⊎` as a destructor, since it plays a similar role. Other terminology refers to `inj₁` and `inj₂` as *introducing* a disjunction, and to `case-⊎` as *eliminating* a disjunction; indeed the former are sometimes given the names `⊎-I₁` and `⊎-I₂` and the latter the name `⊎-E`.

Applying the destructor to each of the constructors is the identity:

```
η-⊎ : ∀ {A B : Set} (w : A ⊎ B) → case-⊎ inj₁ inj₂ w ≡ w
η-⊎ (inj₁ x) = refl
η-⊎ (inj₂ y) = refl
```

More generally, we can also throw in an arbitrary function from a disjunction:

```
uniq-⊎ : ∀ {A B C : Set} (h : A ⊎ B → C) (w : A ⊎ B) →
  case-⊎ (h ∘ inj₁) (h ∘ inj₂) w ≡ h w
uniq-⊎ h (inj₁ x) = refl
uniq-⊎ h (inj₂ y) = refl
```

The pattern matching on the left-hand side is essential. Replacing `w` by `inj₁ x` allows both sides of the propositional equality to simplify to the same term, and similarly for `inj₂ y`.

We set the precedence of disjunction so that it binds less tightly than any other declared operator:

```
infixr 1 _⊎_
```

Thus, `A × C ⊎ B × C` parses as `(A × C) ⊎ (B × C)`.

Given two types `A` and `B`, we refer to `A ⊎ B` as the *sum* of `A` and `B`. In set theory, it is also sometimes called the *disjoint union*, and in computing it corresponds to a *variant record* type. Among other reasons for calling it the sum, note that if type `A` has `m` distinct members, and type `B` has `n` distinct members, then the type `A ⊎ B` has `m + n` distinct members. For instance, consider a type `Bool` with two members, and a type `Tri` with three members, as defined earlier. Then the type `Bool ⊎ Tri` has five members:

```
inj₁ true     inj₂ aa
inj₁ false    inj₂ bb
              inj₂ cc
```

For example, the following function enumerates all possible arguments of type `Bool ⊎ Tri`:

```
⊎-count : Bool ⊎ Tri → ℕ
⊎-count (inj₁ true)  = 1
⊎-count (inj₁ false) = 2
⊎-count (inj₂ aa)    = 3
⊎-count (inj₂ bb)    = 4
⊎-count (inj₂ cc)    = 5
```

Sum on types also shares a property with sum on numbers in that it is commutative and associative *up to isomorphism*.

Exercise `⊎-comm` (recommended)

Show sum is commutative up to isomorphism.

```
-- Your code goes here
```

Exercise `⊎-assoc` (practice)

Show sum is associative up to isomorphism.

```
-- Your code goes here
```

False is empty

False `⊥` never holds. We formalise this idea by declaring a suitable inductive type:

```
data ⊥ : Set where
  -- no clauses!
```

There is no possible evidence that `⊥` holds.

Dual to `⊤`, for `⊥` there is no introduction rule but an elimination rule. Since false never holds, knowing that it holds tells us we are in a paradoxical situation. Given evidence that `⊥` holds, we might conclude anything! This is a basic principle of logic, known in medieval times by the Latin phrase *ex falso*, and known to children through phrases such as "if pigs had wings, then I'd be the Queen of Sheba". We formalise it as follows:

```
⊥-elim : ∀ {A : Set}
  → ⊥
    --
  → A
⊥-elim ()
```

This is our first use of the *absurd pattern* `()`. Here since `⊥` is a type with no members, we indicate that it is *never* possible to match against a value of this type by using the pattern `()`.

The nullary case of `case-⊎` is `⊥-elim`. By analogy, we might have called it `case-⊥`, but chose to stick with the name in the standard library.

The nullary case of `uniq-⊎` is `uniq-⊥`, which asserts that `⊥-elim` is equal to any arbitrary function from `⊥`:

```
uniq-⊥ : ∀ {C : Set} (h : ⊥ → C) (w : ⊥) → ⊥-elim w ≡ h w
uniq-⊥ h ()
```

Using the absurd pattern asserts there are no possible values for `w`, so the equation holds trivially.

We refer to `⊥` as the *empty* type. And, indeed, type `⊥` has no members. For example, the following function enumerates all possible arguments of type `⊥`:

```
⊥-count : ⊥ → ℕ
⊥-count ()
```

Here again the absurd pattern `()` indicates that no value can match type `⊥`.

For numbers, zero is the identity of addition. Correspondingly, empty is the identity of sums *up to isomorphism*.

Exercise `⊥-identityˡ` (recommended)

Show empty is the left identity of sums up to isomorphism.

```
-- Your code goes here
```

Exercise `⊥-identityʳ` (practice)

Show empty is the right identity of sums up to isomorphism.

```
-- Your code goes here
```

Implication is function

Given two propositions `A` and `B`, the implication `A → B` holds if whenever `A` holds then `B` must also hold. We formalise implication using the function type, which has appeared throughout this book.

Evidence that `A → B` holds is of the form

```
λ (x : A) → N
```

where `N` is a term of type `B` containing as a free variable `x` of type `A`. Given a term `L` providing evidence that `A → B` holds, and a term `M` providing evidence that `A` holds, the term `L M` provides evidence that `B` holds. In other words, evidence that `A → B` holds is a function that converts evidence that `A` holds into evidence that `B` holds.

Put another way, if we know that `A → B` and `A` both hold, then we may conclude that `B` holds:

```
→-elim : ∀ {A B : Set}
  → (A → B)
  → A
    -------
  → B
→-elim L M = L M
```

In medieval times, this rule was known by the name *modus ponens*. It corresponds to function application.

Defining a function, with a named definition or a lambda abstraction, is referred to as *introducing* a function, while applying a function is referred to as *eliminating* the function.

Elimination followed by introduction is the identity:

```
η-→ : ∀ {A B : Set} (f : A → B) → (λ (x : A) → f x) ≡ f
η-→ f = refl
```

Implication binds less tightly than any other operator. Thus, `A ⊎ B → B ⊎ A` parses as `(A ⊎ B) → (B ⊎ A)`.

Given two types `A` and `B`, we refer to `A → B` as the *function* space from `A` to `B`. It is also sometimes called the *exponential*, with `B` raised to the `A` power. Among other reasons for calling it the exponential, note that if type `A` has `m` distinct members, and type `B` has `n` distinct members, then the type `A → B` has n^m distinct members. For instance, consider a type `Bool` with two members and a type `Tri` with three members, as defined earlier. Then the type `Bool → Tri` has nine (that is, three squared) members:

```
λ{true → aa; false → aa}  λ{true → aa; false → bb}  λ{true → aa; false → cc}
λ{true → bb; false → aa}  λ{true → bb; false → bb}  λ{true → bb; false → cc}
λ{true → cc; false → aa}  λ{true → cc; false → bb}  λ{true → cc; false → cc}
```

For example, the following function enumerates all possible arguments of the type `Bool → Tri`:

```
→-count : (Bool → Tri) → ℕ
→-count f with f true | f false
... | aa | aa = 1
... | aa | bb = 2
... | aa | cc = 3
... | bb | aa = 4
... | bb | bb = 5
... | bb | cc = 6
... | cc | aa = 7
... | cc | bb = 8
... | cc | cc = 9
```

Exponential on types also share a property with exponential on numbers in that many of the standard identities for numbers carry over to the types.

Corresponding to the law

```
(p ^ n) ^ m  ≡  p ^ (n * m)
```

we have the isomorphism

```
A → (B → C)  ≃  (A × B) → C
```

Both types can be viewed as functions that given evidence that `A` holds and evidence that `B` holds can return evidence that `C` holds. This isomorphism sometimes goes by the name *currying*. The proof of the right inverse requires extensionality:

```
currying : ∀ {A B C : Set} → (A → B → C) ≃ (A × B → C)
currying =
  record
    { to   =   λ{ f → λ{ ⟨ x , y ⟩ → f x y }}
    ; from =   λ{ g → λ{ x → λ{ y → g ⟨ x , y ⟩ }}}
    ; from∘to = λ{ f → refl }
    ; to∘from = λ{ g → extensionality λ{ ⟨ x , y ⟩ → refl }}
    }
```

Currying tells us that instead of a function that takes a pair of arguments, we can have a function that takes the first argument and returns a function that expects the second argument. Thus, for instance, our way of writing addition

```
_+_ : ℕ → ℕ → ℕ
```

is isomorphic to a function that accepts a pair of arguments:

```
_+′_ : (ℕ × ℕ) → ℕ
```

Agda is optimised for currying, so `2 + 3` abbreviates `_+_ 2 3`. In a language optimised for pairing, we would instead take `2 +′ 3` as an abbreviation for `_+′_ ⟨ 2 , 3 ⟩`.

Corresponding to the law

```
p ^ (n + m) = (p ^ n) * (p ^ m)
```

we have the isomorphism:

```
(A ⊎ B) → C  ≃  (A → C) × (B → C)
```

That is, the assertion that if either `A` holds or `B` holds then `C` holds is the same as the assertion that if `A` holds then `C` holds and if `B` holds then `C` holds. The proof of the left inverse requires extensionality:

```
→-distrib-⊎ : ∀ {A B C : Set} → (A ⊎ B → C) ≃ ((A → C) × (B → C))
→-distrib-⊎ =
  record
    { to   = λ{ f → ⟨ f ∘ inj₁ , f ∘ inj₂ ⟩ }
    ; from = λ{ ⟨ g , h ⟩ → λ{ (inj₁ x) → g x ; (inj₂ y) → h y } }
    ; from∘to = λ{ f → extensionality λ{ (inj₁ x) → refl ; (inj₂ y) → refl } }
    ; to∘from = λ{ ⟨ g , h ⟩ → refl }
    }
```

Corresponding to the law

```
(p * n) ^ m = (p ^ m) * (n ^ m)
```

we have the isomorphism:

```
A → B × C  ≃  (A → B) × (A → C)
```

That is, the assertion that if `A` holds then `B` holds and `C` holds is the same as the assertion that if `A` holds then `B` holds and if `A` holds then `C` holds. The proof of left inverse requires both extensionality and the rule `η-×` for products:

```
→-distrib-× : ∀ {A B C : Set} → (A → B × C) ≃ (A → B) × (A → C)
→-distrib-× =
  record
    { to   = λ{ f → ⟨ proj₁ ∘ f , proj₂ ∘ f ⟩ }
    ; from = λ{ ⟨ g , h ⟩ → λ x → ⟨ g x , h x ⟩ }
    ; from∘to = λ{ f → extensionality λ{ x → η-× (f x) } }
    ; to∘from = λ{ ⟨ g , h ⟩ → refl }
    }
```

Distribution

Products distribute over sum, up to isomorphism. The code to validate this fact is similar in structure to our previous results:

```
×-distrib-⊎ : ∀ {A B C : Set} → (A ⊎ B) × C ≃ (A × C) ⊎ (B × C)
×-distrib-⊎ =
  record
    { to   = λ{ ⟨ inj₁ x , z ⟩ → (inj₁ ⟨ x , z ⟩)
              ; ⟨ inj₂ y , z ⟩ → (inj₂ ⟨ y , z ⟩)
              }
    ; from = λ{ (inj₁ ⟨ x , z ⟩) → ⟨ inj₁ x , z ⟩
              ; (inj₂ ⟨ y , z ⟩) → ⟨ inj₂ y , z ⟩
              }
    ; from∘to = λ{ ⟨ inj₁ x , z ⟩ → refl
              ; ⟨ inj₂ y , z ⟩ → refl
              }
    ; to∘from = λ{ (inj₁ ⟨ x , z ⟩) → refl
              ; (inj₂ ⟨ y , z ⟩) → refl
              }
    }
```

Sums do not distribute over products up to isomorphism, but it is an embedding:

```
⊎-distrib-× : ∀ {A B C : Set} → (A × B) ⊎ C ≲ (A ⊎ C) × (B ⊎ C)
⊎-distrib-× =
  record
    { to   = λ{ (inj₁ ⟨ x , y ⟩) → ⟨ inj₁ x , inj₁ y ⟩
              ; (inj₂ z) → ⟨ inj₂ z , inj₂ z ⟩
              }
    ; from = λ{ ⟨ inj₁ x , inj₁ y ⟩ → (inj₁ ⟨ x , y ⟩)
              ; ⟨ inj₁ x , inj₂ z ⟩ → (inj₂ z)
              ; ⟨ inj₂ z , _ ⟩ → (inj₂ z)
              }
    ; from∘to = λ{ (inj₁ ⟨ x , y ⟩) → refl
              ; (inj₂ z) → refl
              }
    }
```

Note that there is a choice in how we write the `from` function. As given, it takes `⟨ inj₂ z , inj₂ z′ ⟩` to `inj₂ z`, but it is easy to write a variant that instead returns `inj₂ z′`. We have an embedding rather than an isomorphism because the `from` function must discard

either `z` or `z′` in this case.

In the usual approach to logic, both of the distribution laws are given as equivalences, where each side implies the other:

```
A × (B ⊎ C) ⇔ (A × B) ⊎ (A × C)
A ⊎ (B × C) ⇔ (A ⊎ B) × (A ⊎ C)
```

But when we consider the functions that provide evidence for these implications, then the first corresponds to an isomorphism while the second only corresponds to an embedding, revealing a sense in which one of these laws is "more true" than the other.

Exercise `⊎-weak-×` (recommended)

Show that the following property holds:

```
postulate
  ⊎-weak-× : ∀ {A B C : Set} → (A ⊎ B) × C → A ⊎ (B × C)
```

This is called a *weak distributive law*. Give the corresponding distributive law, and explain how it relates to the weak version.

```
-- Your code goes here
```

Exercise `⊎×-implies-×⊎` (practice)

Show that a disjunct of conjuncts implies a conjunct of disjuncts:

```
postulate
  ⊎×-implies-×⊎ : ∀ {A B C D : Set} → (A × B) ⊎ (C × D) → (A ⊎ C) × (B ⊎ D)
```

Does the converse hold? If so, prove; if not, give a counterexample.

```
-- Your code goes here
```

Standard library

Definitions similar to those in this chapter can be found in the standard library:

```
import Data.Product using (_×_; proj₁; proj₂) renaming (_,_ to ⟨_,_⟩)
import Data.Unit using (⊤; tt)
import Data.Sum using (_⊎_; inj₁; inj₂) renaming ([_,_] to case-⊎)
import Data.Empty using (⊥; ⊥-elim)
import Function.Equivalence using (_⇔_)
```

The standard library constructs pairs with `_,_` whereas we use `⟨_,_⟩`. The former makes it convenient to build triples or larger tuples from pairs, permitting `a , b , c` to stand for `(a , (b , c))`. But it conflicts with other useful notations, such as `[_,_]` to construct a list of two elements in Chapter Lists and `Γ , A` to extend environments in Chapter DeBruijn. The

standard library `_⇔_` is similar to ours, but the one in the standard library is less convenient, since it is parameterised with respect to an arbitrary notion of equivalence.

Unicode

This chapter uses the following unicode:

```
×  U+00D7  MULTIPLICATION SIGN (\x)
⊎  U+228E  MULTISET UNION (\u+)
⊤  U+22A4  DOWN TACK (\top)
⊥  U+22A5  UP TACK (\bot)
η  U+03B7  GREEK SMALL LETTER ETA (\eta)
₁  U+2081  SUBSCRIPT ONE (\_1)
₂  U+2082  SUBSCRIPT TWO (\_2)
⇔  U+21D4  LEFT RIGHT DOUBLE ARROW (\<=>)
```

Chapter 7

Negation: Negation, with intuitionistic and classical logic

```
module plfa.part1.Negation where
```

This chapter introduces negation, and discusses intuitionistic and classical logic.

Imports

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Data.Nat using (ℕ; zero; suc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Data.Product using (_×_)
open import plfa.part1.Isomorphism using (_≃_; extensionality)
```

Negation

Given a proposition `A`, the negation `¬ A` holds if `A` cannot hold. We formalise this idea by declaring negation to be the same as implication of false:

```
¬_ : Set → Set
¬ A = A → ⊥
```

This is a form of *reductio ad absurdum*: if assuming `A` leads to the conclusion `⊥` (an absurdity), then we must have `¬ A`.

Evidence that `¬ A` holds is of the form

```
λ{ x → N }
```

where `N` is a term of type `⊥` containing as a free variable `x` of type `A`. In other words, evidence that `¬ A` holds is a function that converts evidence that `A` holds into evidence that `⊥` holds.

Given evidence that both `¬ A` and `A` hold, we can conclude that `⊥` holds. In other words, if both `¬ A` and `A` hold, then we have a contradiction:

```
¬-elim : ∀ {A : Set}
  → ¬ A
  → A
    ---
  → ⊥
¬-elim ¬x x = ¬x x
```

Here we write `¬x` for evidence of `¬ A` and `x` for evidence of `A`. This means that `¬x` must be a function of type `A → ⊥`, and hence the application `¬x x` must be of type `⊥`. Note that this rule is just a special case of `→-elim`.

We set the precedence of negation so that it binds more tightly than disjunction and conjunction, but less tightly than anything else:

```
infix 3 ¬_
```

Thus, `¬ A × ¬ B` parses as `(¬ A) × (¬ B)` and `¬ m ≡ n` as `¬ (m ≡ n)`.

In *classical* logic, we have that `A` is equivalent to `¬ ¬ A`. As we discuss below, in Agda we use *intuitionistic* logic, where we have only half of this equivalence, namely that `A` implies `¬ ¬ A`:

```
¬¬-intro : ∀ {A : Set}
  → A
    -----
  → ¬ ¬ A
¬¬-intro x = λ{¬x → ¬x x}
```

Let `x` be evidence of `A`. We show that assuming `¬ A` leads to a contradiction, and hence `¬ ¬ A` must hold. Let `¬x` be evidence of `¬ A`. Then from `A` and `¬ A` we have a contradiction, evidenced by `¬x x`. Hence, we have shown `¬ ¬ A`.

An equivalent way to write the above is as follows:

```
¬¬-intro′ : ∀ {A : Set}
  → A
    -----
  → ¬ ¬ A
¬¬-intro′ x ¬x = ¬x x
```

Here we have simply converted the argument of the lambda term to an additional argument of the function. We will usually use this latter style, as it is more compact.

We cannot show that `¬ ¬ A` implies `A`, but we can show that `¬ ¬ ¬ A` implies `¬ A`:

```
¬¬¬-elim : ∀ {A : Set}
  → ¬ ¬ ¬ A
    -------
  → ¬ A
¬¬¬-elim ¬¬¬x = λ x → ¬¬¬x (¬¬-intro x)
```

Let `¬¬¬x` be evidence of `¬ ¬ ¬ A`. We will show that assuming `A` leads to a contradiction, and hence `¬ A` must hold. Let `x` be evidence of `A`. Then by the previous result, we can conclude

`¬ ¬ A`, evidenced by `¬¬-intro x`. Then from `¬ ¬ ¬ A` and `¬ ¬ A` we have a contradiction, evidenced by `¬¬¬x (¬¬-intro x)`. Hence we have shown `¬ A`.

Another law of logic is *contraposition*, stating that if `A` implies `B`, then `¬ B` implies `¬ A`:

```
contraposition : ∀ {A B : Set}
  → (A → B)
    -----------
  → (¬ B → ¬ A)
contraposition f ¬y x = ¬y (f x)
```

Let `f` be evidence of `A → B` and let `¬y` be evidence of `¬ B`. We will show that assuming `A` leads to a contradiction, and hence `¬ A` must hold. Let `x` be evidence of `A`. Then from `A → B` and `A` we may conclude `B`, evidenced by `f x`, and from `B` and `¬ B` we may conclude `⊥`, evidenced by `¬y (f x)`. Hence, we have shown `¬ A`.

Using negation, it is straightforward to define inequality:

```
_≢_ : ∀ {A : Set} → A → A → Set
x ≢ y = ¬ (x ≡ y)
```

It is trivial to show distinct numbers are not equal:

```
_ : 1 ≢ 2
_ = λ()
```

This is our first use of an absurd pattern in a lambda expression. The type `M ≡ N` is occupied exactly when `M` and `N` simplify to identical terms. Since `1` and `2` simplify to distinct normal forms, Agda determines that there is no possible evidence that `1 ≡ 2`. As a second example, it is also easy to validate Peano's postulate that zero is not the successor of any number:

```
peano : ∀ {m : ℕ} → zero ≢ suc m
peano = λ()
```

The evidence is essentially the same, as the absurd pattern matches all possible evidence of type `zero ≡ suc m`.

Given the correspondence of implication to exponentiation and false to the type with no members, we can view negation as raising to the zero power. This indeed corresponds to what we know for arithmetic, where

```
0 ^ n  ≡  1,  if n ≡ 0
       ≡  0,  if n ≢ 0
```

Indeed, there is exactly one proof of `⊥ → ⊥`. We can write this proof two different ways:

```
id : ⊥ → ⊥
id x = x

id′ : ⊥ → ⊥
id′ ()
```

But, using extensionality, we can prove these equal:

```
id≡id′ : id ≡ id′
id≡id′ = extensionality (λ())
```

By extensionality, `id ≡ id′` holds if for every `x` in their domain we have `id x ≡ id′ x`. But there is no `x` in their domain, so the equality holds trivially.

Indeed, we can show any two proofs of a negation are equal:

```
assimilation : ∀ {A : Set} (¬x ¬x′ : ¬ A) → ¬x ≡ ¬x′
assimilation ¬x ¬x′ = extensionality (λ x → ⊥-elim (¬x x))
```

Evidence for `¬ A` implies that any evidence of `A` immediately leads to a contradiction. But extensionality quantifies over all `x` such that `A` holds, hence any such `x` immediately leads to a contradiction, again causing the equality to hold trivially.

Exercise `<-irreflexive` (recommended)

Using negation, show that strict inequality is irreflexive, that is, `n < n` holds for no `n`.

```
-- Your code goes here
```

Exercise `trichotomy` (practice)

Show that strict inequality satisfies trichotomy, that is, for any naturals `m` and `n` exactly one of the following holds:

- `m < n`
- `m ≡ n`
- `m > n`

Here "exactly one" means that not only one of the three must hold, but that when one holds the negation of the other two must also hold.

```
-- Your code goes here
```

Exercise `⊎-dual-×` (recommended)

Show that conjunction, disjunction, and negation are related by a version of De Morgan's Law.

```
¬ (A ⊎ B) ≃ (¬ A) × (¬ B)
```

This result is an easy consequence of something we've proved previously.

```
-- Your code goes here
```

Do we also have the following?

```
¬ (A × B) ≃ (¬ A) ⊎ (¬ B)
```

If so, prove; if not, can you give a relation weaker than isomorphism that relates the two sides?

Intuitive and Classical logic

In Gilbert and Sullivan's *The Gondoliers*, Casilda is told that as an infant she was married to the heir of the King of Batavia, but that due to a mix-up no one knows which of two individuals, Marco or Giuseppe, is the heir. Alarmed, she wails "Then do you mean to say that I am married to one of two gondoliers, but it is impossible to say which?" To which the response is "Without any doubt of any kind whatever."

Logic comes in many varieties, and one distinction is between *classical* and *intuitionistic*. Intuitionists, concerned by assumptions made by some logicians about the nature of infinity, insist upon a constructionist notion of truth. In particular, they insist that a proof of `A ⊎ B` must show *which* of `A` or `B` holds, and hence they would reject the claim that Casilda is married to Marco or Giuseppe until one of the two was identified as her husband. Perhaps Gilbert and Sullivan anticipated intuitionism, for their story's outcome is that the heir turns out to be a third individual, Luiz, with whom Casilda is, conveniently, already in love.

Intuitionists also reject the law of the excluded middle, which asserts `A ⊎ ¬ A` for every `A`, since the law gives no clue as to *which* of `A` or `¬ A` holds. Heyting formalised a variant of Hilbert's classical logic that captures the intuitionistic notion of provability. In particular, the law of the excluded middle is provable in Hilbert's logic, but not in Heyting's. Further, if the law of the excluded middle is added as an axiom to Heyting's logic, then it becomes equivalent to Hilbert's. Kolmogorov showed the two logics were closely related: he gave a double-negation translation, such that a formula is provable in classical logic if and only if its translation is provable in intuitionistic logic.

Propositions as Types was first formulated for intuitionistic logic. It is a perfect fit, because in the intuitionist interpretation the formula `A ⊎ B` is provable exactly when one exhibits either a proof of `A` or a proof of `B`, so the type corresponding to disjunction is a disjoint sum.

(Parts of the above are adopted from "Propositions as Types", Philip Wadler, *Communications of the ACM*, December 2015.)

Excluded middle is irrefutable

The law of the excluded middle can be formulated as follows:

```
postulate
  em : ∀ {A : Set} → A ⊎ ¬ A
```

As we noted, the law of the excluded middle does not hold in intuitionistic logic. However, we can show that it is *irrefutable*, meaning that the negation of its negation is provable (and hence that its negation is never provable):

```
em-irrefutable : ∀ {A : Set} → ¬ ¬ (A ⊎ ¬ A)
em-irrefutable = λ k → k (inj₂ (λ x → k (inj₁ x)))
```

The best way to explain this code is to develop it interactively:

```
em-irrefutable k = ?
```

Given evidence `k` that `¬ (A ⊎ ¬ A)`, that is, a function that given a value of type `A ⊎ ¬ A` returns a value of the empty type, we must fill in `?` with a term that returns a value of the empty type. The only way we can get a value of the empty type is by applying `k` itself, so let's expand the hole accordingly:

```
em-irrefutable k = k ?
```

We need to fill the new hole with a value of type `A ⊎ ¬ A`. We don't have a value of type `A` to hand, so let's pick the second disjunct:

```
em-irrefutable k = k (inj₂ λ{ x → ? })
```

The second disjunct accepts evidence of `¬ A`, that is, a function that given a value of type `A` returns a value of the empty type. We bind `x` to the value of type `A`, and now we need to fill in the hole with a value of the empty type. Once again, the only way we can get a value of the empty type is by applying `k` itself, so let's expand the hole accordingly:

```
em-irrefutable k = k (inj₂ λ{ x → k ? })
```

This time we do have a value of type `A` to hand, namely `x`, so we can pick the first disjunct:

```
em-irrefutable k = k (inj₂ λ{ x → k (inj₁ x) })
```

There are no holes left! This completes the proof.

The following story illustrates the behaviour of the term we have created. (With apologies to Peter Selinger, who tells a similar story about a king, a wizard, and the Philosopher's stone.)

Once upon a time, the devil approached a man and made an offer: "Either (a) I will give you one billion dollars, or (b) I will grant you any wish if you pay me one billion dollars. Of course, I get to choose whether I offer (a) or (b)."

The man was wary. Did he need to sign over his soul? No, said the devil, all the man need do is accept the offer.

The man pondered. If he was offered (b) it was unlikely that he would ever be able to buy the wish, but what was the harm in having the opportunity available?

"I accept," said the man at last. "Do I get (a) or (b)?"

The devil paused. "I choose (b)."

The man was disappointed but not surprised. That was that, he thought. But the offer gnawed at him. Imagine what he could do with his wish! Many years passed, and the man began to accumulate money. To get the money he sometimes did bad things, and dimly he realised that this must be what the devil had in mind. Eventually he had his billion dollars, and the devil appeared again.

"Here is a billion dollars," said the man, handing over a valise containing the money. "Grant me my wish!"

The devil took possession of the valise. Then he said, "Oh, did I say (b) before? I'm so sorry. I meant (a). It is my great pleasure to give you one billion dollars."

And the devil handed back to the man the same valise that the man had just handed to him.

(Parts of the above are adopted from "Call-by-Value is Dual to Call-by-Name", Philip Wadler, *International Conference on Functional Programming*, 2003.)

Exercise `Classical` (stretch)

Consider the following principles:

- Excluded Middle: `A ⊎ ¬ A`, for all `A`
- Double Negation Elimination: `¬ ¬ A → A`, for all `A`
- Peirce's Law: `((A → B) → A) → A`, for all `A` and `B`.
- Implication as disjunction: `(A → B) → ¬ A ⊎ B`, for all `A` and `B`.
- De Morgan: `¬ (¬ A × ¬ B) → A ⊎ B`, for all `A` and `B`.

Show that each of these implies all the others.

```
-- Your code goes here
```

Exercise `Stable` (stretch)

Say that a formula is *stable* if double negation elimination holds for it:

```
Stable : Set → Set
Stable A = ¬ ¬ A → A
```

Show that any negated formula is stable, and that the conjunction of two stable formulas is stable.

```
-- Your code goes here
```

Standard Prelude

Definitions similar to those in this chapter can be found in the standard library:

```
import Relation.Nullary using (¬_)
import Relation.Nullary.Negation using (contraposition)
```

Unicode

This chapter uses the following unicode:

```
¬  U+00AC  NOT SIGN (\neg)
≢  U+2262  NOT IDENTICAL TO (\==n)
```

Chapter 8

Quantifiers: Universals and existentials

```
module plfa.part1.Quantifiers where
```

This chapter introduces universal and existential quantification.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Nat using (ℕ; zero; suc; _+_; _*_)
open import Relation.Nullary using (¬_)
open import Data.Product using (_×_; proj₁; proj₂) renaming (_,_ to ⟨_,_⟩)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import plfa.part1.Isomorphism using (_≃_; extensionality)
```

Universals

We formalise universal quantification using the dependent function type, which has appeared throughout this book. For instance, in Chapter Induction we showed addition is associative:

```
+-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
```

which asserts for all natural numbers `m`, `n`, and `p` that `(m + n) + p ≡ m + (n + p)` holds. It is a dependent function, which given values for `m`, `n`, and `p` returns evidence for the corresponding equation.

In general, given a variable `x` of type `A` and a proposition `B x` which contains `x` as a free variable, the universally quantified proposition `∀ (x : A) → B x` holds if for every term `M` of type `A` the proposition `B M` holds. Here `B M` stands for the proposition `B x` with each free occurrence of `x` replaced by `M`. Variable `x` appears free in `B x` but bound in `∀ (x : A) → B x`.

Evidence that `∀ (x : A) → B x` holds is of the form

```
λ (x : A) → N x
```

where `N x` is a term of type `B x`, and `N x` and `B x` both contain a free variable `x` of type `A`. Given a term `L` providing evidence that `∀ (x : A) → B x` holds, and a term `M` of type `A`, the term `L M` provides evidence that `B M` holds. In other words, evidence that `∀ (x : A) → B x` holds is a function that converts a term `M` of type `A` into evidence that `B M` holds.

Put another way, if we know that `∀ (x : A) → B x` holds and that `M` is a term of type `A` then we may conclude that `B M` holds:

```
∀-elim : ∀ {A : Set} {B : A → Set}
  → (L : ∀ (x : A) → B x)
  → (M : A)
    -----------------
  → B M
∀-elim L M = L M
```

As with `→-elim`, the rule corresponds to function application.

Functions arise as a special case of dependent functions, where the range does not depend on a variable drawn from the domain. When a function is viewed as evidence of implication, both its argument and result are viewed as evidence, whereas when a dependent function is viewed as evidence of a universal, its argument is viewed as an element of a data type and its result is viewed as evidence of a proposition that depends on the argument. This difference is largely a matter of interpretation, since in Agda a value of a type and evidence of a proposition are indistinguishable.

Dependent function types are sometimes referred to as dependent products, because if `A` is a finite type with values x_1 , ⋯ , x_n, and if each of the types `B` x_1 , ⋯ , `B` x_n has m_1 , ⋯ , m_n distinct members, then `∀ (x : A) → B x` has $m_1 * \cdots * m_n$ members. Indeed, sometimes the notation `∀ (x : A) → B x` is replaced by a notation such as `Π[ x ∈ A ] (B x)`, where `Π` stands for product. However, we will stick with the name dependent function, because (as we will see) dependent product is ambiguous.

Exercise `∀-distrib-×` (recommended)

Show that universals distribute over conjunction:

```
postulate
  ∀-distrib-× : ∀ {A : Set} {B C : A → Set} →
    (∀ (x : A) → B x × C x) ≃ (∀ (x : A) → B x) × (∀ (x : A) → C x)
```

Compare this with the result (`→-distrib-×`) in Chapter Connectives.

Exercise `⊎∀-implies-∀⊎` (practice)

Show that a disjunction of universals implies a universal of disjunctions:

```
postulate
  ⊎∀-implies-∀⊎ : ∀ {A : Set} {B C : A → Set} →
    (∀ (x : A) → B x) ⊎ (∀ (x : A) → C x) → ∀ (x : A) → B x ⊎ C x
```

Does the converse hold? If so, prove; if not, explain why.

Exercise `∀-×` (practice)

Consider the following type.

```
data Tri : Set where
  aa : Tri
  bb : Tri
  cc : Tri
```

Let `B` be a type indexed by `Tri`, that is `B : Tri → Set`. Show that `∀ (x : Tri) → B x` is isomorphic to `B aa × B bb × B cc`. Hint: you will need to postulate a version of extensionality that works for dependent functions.

Existentials

Given a variable `x` of type `A` and a proposition `B x` which contains `x` as a free variable, the existentially quantified proposition `Σ[ x ∈ A ] B x` holds if for some term `M` of type `A` the proposition `B M` holds. Here `B M` stands for the proposition `B x` with each free occurrence of `x` replaced by `M`. Variable `x` appears free in `B x` but bound in `Σ[ x ∈ A ] B x`.

We formalise existential quantification by declaring a suitable inductive type:

```
data Σ (A : Set) (B : A → Set) : Set where
  ⟨_,_⟩ : (x : A) → B x → Σ A B
```

We define a convenient syntax for existentials as follows:

```
Σ-syntax = Σ
infix 2 Σ-syntax
syntax Σ-syntax A (λ x → B) = Σ[ x ∈ A ] B
```

This is our first use of a syntax declaration, which specifies that the term on the left may be written with the syntax on the right. The special syntax is available only when the identifier `Σ-syntax` is imported.

Evidence that `Σ[ x ∈ A ] B x` holds is of the form `⟨ M , N ⟩` where `M` is a term of type `A`, and `N` is evidence that `B M` holds.

Equivalently, we could also declare existentials as a record type:

```
record Σ′ (A : Set) (B : A → Set) : Set where
  field
    proj₁′ : A
    proj₂′ : B proj₁′
```

Here record construction

```
record
  { proj₁′ = M
  ; proj₂′ = N
```

```
  }
```

corresponds to the term

```
⟨ M , N ⟩
```

where `M` is a term of type `A` and `N` is a term of type `B M`.

Products arise as a special case of existentials, where the second component does not depend on a variable drawn from the first component. When a product is viewed as evidence of a conjunction, both of its components are viewed as evidence, whereas when it is viewed as evidence of an existential, the first component is viewed as an element of a datatype and the second component is viewed as evidence of a proposition that depends on the first component. This difference is largely a matter of interpretation, since in Agda a value of a type and evidence of a proposition are indistinguishable.

Existentials are sometimes referred to as dependent sums, because if `A` is a finite type with values `x₁ , ⋯ , xₙ`, and if each of the types `B x₁ , ⋯ B xₙ` has `m₁ , ⋯ , mₙ` distinct members, then `Σ[ x ∈ A ] B x` has `m₁ + ⋯ + mₙ` members, which explains the choice of notation for existentials, since `Σ` stands for sum.

Existentials are sometimes referred to as dependent products, since products arise as a special case. However, that choice of names is doubly confusing, since universals also have a claim to the name dependent product and since existentials also have a claim to the name dependent sum.

A common notation for existentials is `∃` (analogous to `∀` for universals). We follow the convention of the Agda standard library, and reserve this notation for the case where the domain of the bound variable is left implicit:

```
∃ : ∀ {A : Set} (B : A → Set) → Set
∃ {A} B = Σ A B

∃-syntax = ∃
syntax ∃-syntax (λ x → B) = ∃[ x ] B
```

The special syntax is available only when the identifier `∃-syntax` is imported. We will tend to use this syntax, since it is shorter and more familiar.

Given evidence that `∀ x → B x → C` holds, where `C` does not contain `x` as a free variable, and given evidence that `∃[ x ] B x` holds, we may conclude that `C` holds:

```
∃-elim : ∀ {A : Set} {B : A → Set} {C : Set}
  → (∀ x → B x → C)
  → ∃[ x ] B x
    ---------------
  → C
∃-elim f ⟨ x , y ⟩ = f x y
```

In other words, if we know for every `x` of type `A` that `B x` implies `C`, and we know for some `x` of type `A` that `B x` holds, then we may conclude that `C` holds. This is because we may instantiate that proof that `∀ x → B x → C` to any value `x` of type `A` and any `y` of type `B x`, and exactly such values are provided by the evidence for `∃[ x ] B x`.

Indeed, the converse also holds, and the two together form an isomorphism:

```
∀∃-currying : ∀ {A : Set} {B : A → Set} {C : Set}
  → (∀ x → B x → C) ≃ (∃[ x ] B x → C)
∀∃-currying =
  record
    { to   =   λ{ f → λ{ ⟨ x , y ⟩ → f x y }}
    ; from =   λ{ g → λ{ x → λ{ y → g ⟨ x , y ⟩ }}}
    ; from∘to = λ{ f → refl }
    ; to∘from = λ{ g → extensionality λ{ ⟨ x , y ⟩ → refl }}
    }
```

The result can be viewed as a generalisation of currying. Indeed, the code to establish the isomorphism is identical to what we wrote when discussing implication.

Exercise `∃-distrib-⊎` (recommended)

Show that existentials distribute over disjunction:

```
postulate
  ∃-distrib-⊎ : ∀ {A : Set} {B C : A → Set} →
    ∃[ x ] (B x ⊎ C x) ≃ (∃[ x ] B x) ⊎ (∃[ x ] C x)
```

Exercise `∃×-implies-×∃` (practice)

Show that an existential of conjunctions implies a conjunction of existentials:

```
postulate
  ∃×-implies-×∃ : ∀ {A : Set} {B C : A → Set} →
    ∃[ x ] (B x × C x) → (∃[ x ] B x) × (∃[ x ] C x)
```

Does the converse hold? If so, prove; if not, explain why.

Exercise `∃-⊎` (practice)

Let `Tri` and `B` be as in Exercise `∀-×`. Show that `∃[ x ] B x` is isomorphic to `B aa ⊎ B bb ⊎ B cc`.

An existential example

Recall the definitions of `even` and `odd` from Chapter Relations:

```
data even : ℕ → Set
data odd : ℕ → Set

data even where

  even-zero : even zero

  even-suc : ∀ {n : ℕ}
    → odd n
```

```
    ------------
  → even (suc n)

data odd where
  odd-suc : ∀ {n : ℕ}
    → even n
      -----------
    → odd (suc n)
```

A number is even if it is zero or the successor of an odd number, and odd if it is the successor of an even number.

We will show that a number is even if and only if it is twice some other number, and odd if and only if it is one more than twice some other number. In other words, we will show:

`even n` iff `∃[ m ] (    m * 2 ≡ n)`

`odd  n` iff `∃[ m ] (1 + m * 2 ≡ n)`

By convention, one tends to write constant factors first and to put the constant term in a sum last. Here we've reversed each of those conventions, because doing so eases the proof.

Here is the proof in the forward direction:

```
even-∃ : ∀ {n : ℕ} → even n → ∃[ m ] ( m * 2 ≡ n)
odd-∃ : ∀ {n : ℕ} → odd n → ∃[ m ] (1 + m * 2 ≡ n)

even-∃ even-zero =                 ⟨ zero , refl ⟩
even-∃ (even-suc o) with odd-∃ o
... | ⟨ m , refl ⟩ =               ⟨ suc m , refl ⟩

odd-∃ (odd-suc e) with even-∃ e
... | ⟨ m , refl ⟩ =               ⟨ m , refl ⟩
```

We define two mutually recursive functions. Given evidence that `n` is even or odd, we return a number `m` and evidence that `m * 2 ≡ n` or `1 + m * 2 ≡ n`. We induct over the evidence that `n` is even or odd:

- If the number is even because it is zero, then we return a pair consisting of zero and the evidence that twice zero is zero.
- If the number is even because it is one more than an odd number, then we apply the induction hypothesis to give a number `m` and evidence that `1 + m * 2 ≡ n`. We return a pair consisting of `suc m` and evidence that `suc m * 2 ≡ suc n`, which is immediate after substituting for `n`.
- If the number is odd because it is the successor of an even number, then we apply the induction hypothesis to give a number `m` and evidence that `m * 2 ≡ n`. We return a pair consisting of `suc m` and evidence that `1 + m * 2 ≡ suc n`, which is immediate after substituting for `n`.

This completes the proof in the forward direction.

Here is the proof in the reverse direction:

```
∃-even : ∀ {n : ℕ} → ∃[ m ] ( m * 2 ≡ n) → even n
∃-odd : ∀ {n : ℕ} → ∃[ m ] (1 + m * 2 ≡ n) → odd n

∃-even ⟨ zero , refl ⟩ = even-zero
```

```
∃-even ⟨ suc m , refl ⟩ = even-suc (∃-odd ⟨ m , refl ⟩)

∃-odd ⟨ m , refl ⟩      = odd-suc (∃-even ⟨ m , refl ⟩)
```

Given a number that is twice some other number we must show it is even, and a number that is one more than twice some other number we must show it is odd. We induct over the evidence of the existential, and in the even case consider the two possibilities for the number that is doubled:

- In the even case for `zero`, we must show `zero * 2` is even, which follows by `even-zero`.
- In the even case for `suc n`, we must show `suc m * 2` is even. The inductive hypothesis tells us that `1 + m * 2` is odd, from which the desired result follows by `even-suc`.
- In the odd case, we must show `1 + m * 2` is odd. The inductive hypothesis tell us that `m * 2` is even, from which the desired result follows by `odd-suc`.

This completes the proof in the backward direction.

Exercise `∃-even-odd` (practice)

How do the proofs become more difficult if we replace `m * 2` and `1 + m * 2` by `2 * m` and `2 * m + 1`? Rewrite the proofs of `∃-even` and `∃-odd` when restated in this way.

```
-- Your code goes here
```

Exercise `∃-|-≤` (practice)

Show that `y ≤ z` holds if and only if there exists a `x` such that `x + y ≡ z`.

```
-- Your code goes here
```

Existentials, Universals, and Negation

Negation of an existential is isomorphic to the universal of a negation. Considering that existentials are generalised disjunction and universals are generalised conjunction, this result is analogous to the one which tells us that negation of a disjunction is isomorphic to a conjunction of negations:

```
¬∃≃∀¬ : ∀ {A : Set} {B : A → Set}
  → (¬ ∃[ x ] B x) ≃ ∀ x → ¬ B x
¬∃≃∀¬ =
  record
    { to   =    λ{ ¬∃xy x y → ¬∃xy ⟨ x , y ⟩ }
    ; from =    λ{ ∀¬xy ⟨ x , y ⟩ → ∀¬xy x y }
    ; from∘to = λ{ ¬∃xy → extensionality λ{ ⟨ x , y ⟩ → refl } }
    ; to∘from = λ{ ∀¬xy → refl }
    }
```

In the `to` direction, we are given a value `¬∃xy` of type `¬ ∃[ x ] B x`, and need to show that given a value `x` that `¬ B x` follows, in other words, from a value `y` of type `B x` we can derive false. Combining `x` and `y` gives us a value `⟨ x , y ⟩` of type `∃[ x ] B x`, and applying `¬∃xy` to that yields a contradiction.

In the `from` direction, we are given a value `∀¬xy` of type `∀ x → ¬ B x`, and need to show that from a value `⟨ x , y ⟩` of type `∃[ x ] B x` we can derive false. Applying `∀¬xy` to `x` gives a value of type `¬ B x`, and applying that to `y` yields a contradiction.

The two inverse proofs are straightforward, where one direction requires extensionality.

Exercise `∃¬-implies-¬∀` (recommended)

Show that existential of a negation implies negation of a universal:

```
postulate
  ∃¬-implies-¬∀ : ∀ {A : Set} {B : A → Set}
    → ∃[ x ] (¬ B x)
      --------------
    → ¬ (∀ x → B x)
```

Does the converse hold? If so, prove; if not, explain why.

Exercise `Bin-isomorphism` (stretch)

Recall that Exercises Bin, Bin-laws, and Bin-predicates define a datatype `Bin` of bitstrings representing natural numbers, and asks you to define the following functions and predicates:

```
to   : ℕ → Bin
from : Bin → ℕ
Can  : Bin → Set
```

And to establish the following properties:

```
from (to n) ≡ n

----------
Can (to n)

Can b
---------------
to (from b) ≡ b
```

Using the above, establish that there is an isomorphism between `ℕ` and `∃[ b ] Can b`.

We recommend proving the following lemmas which show that, for a given binary number `b`, there is only one proof of `One b` and similarly for `Can b`.

```
≡One : ∀ {b : Bin} (o o′ : One b) → o ≡ o′

≡Can : ∀ {b : Bin} (cb cb′ : Can b) → cb ≡ cb′
```

Many of the alternatives for proving `to∘from` turn out to be tricky. However, the proof can be straightforward if you use the following lemma, which is a corollary of `≡Can`.

```
proj₁≡→Can≡ : {cb cb′ : ∃[ b ] Can b} → proj₁ cb ≡ proj₁ cb′ → cb ≡ cb′
```

```
-- Your code goes here
```

Standard library

Definitions similar to those in this chapter can be found in the standard library:

```
import Data.Product using (Σ; _,_; ∃; Σ-syntax; ∃-syntax)
```

Unicode

This chapter uses the following unicode:

```
Π  U+03A0  GREEK CAPITAL LETTER PI (\Pi)
Σ  U+03A3  GREEK CAPITAL LETTER SIGMA (\Sigma)
∃  U+2203  THERE EXISTS (\ex, \exists)
```

Chapter 9

Decidable: Booleans and decision procedures

```
module plfa.part1.Decidable where
```

We have a choice as to how to represent relations: as an inductive data type of *evidence* that the relation holds, or as a function that *computes* whether the relation holds. Here we explore the relation between these choices. We first explore the familiar notion of *booleans*, but later discover that these are best avoided in favour of a new notion of *decidable*.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open Eq.≡-Reasoning
open import Data.Nat using (ℕ; zero; suc)
open import Data.Product using (_×_) renaming (_,_ to ⟨_,_⟩)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Negation using ()
  renaming (contradiction to ¬¬-intro)
open import Data.Unit using (⊤; tt)
open import Data.Empty using (⊥; ⊥-elim)
open import plfa.part1.Relations using (_<_; z<s; s<s)
open import plfa.part1.Isomorphism using (_⇔_)
```

Evidence vs Computation

Recall that Chapter Relations defined comparison as an inductive datatype, which provides *evidence* that one number is less than or equal to another:

```
infix 4 _≤_

data _≤_ : ℕ → ℕ → Set where
```

```
  z≤n : ∀ {n : ℕ}
      --------
    → zero ≤ n

  s≤s : ∀ {m n : ℕ}
    → m ≤ n
      -------------
    → suc m ≤ suc n
```

For example, we can provide evidence that `2 ≤ 4`, and show there is no possible evidence that `4 ≤ 2`:

```
2≤4 : 2 ≤ 4
2≤4 = s≤s (s≤s z≤n)

¬4≤2 : ¬ (4 ≤ 2)
¬4≤2 (s≤s (s≤s ()))
```

The occurrence of `()` attests to the fact that there is no possible evidence for `2 ≤ 0`, which `z≤n` cannot match (because `2` is not `zero`) and `s≤s` cannot match (because `0` cannot match `suc n`).

An alternative, which may seem more familiar, is to define a type of booleans:

```
data Bool : Set where
  true : Bool
  false : Bool
```

Given booleans, we can define a function of two numbers that *computes* to `true` if the comparison holds and to `false` otherwise:

```
infix 4 _≤ᵇ_

_≤ᵇ_ : ℕ → ℕ → Bool
zero ≤ᵇ n     = true
suc m ≤ᵇ zero  = false
suc m ≤ᵇ suc n = m ≤ᵇ n
```

The first and last clauses of this definition resemble the two constructors of the corresponding inductive datatype, while the middle clause arises because there is no possible evidence that `suc m ≤ zero` for any `m`. For example, we can compute that `2 ≤ᵇ 4` holds, and we can compute that `4 ≤ᵇ 2` does not hold:

```
_ : (2 ≤ᵇ 4) ≡ true
_ =
  begin
    2 ≤ᵇ 4
  ≡⟨⟩
    1 ≤ᵇ 3
  ≡⟨⟩
    0 ≤ᵇ 2
  ≡⟨⟩
    true
  ∎

_ : (4 ≤ᵇ 2) ≡ false
_ =
```

```
begin
  4 ≤ᵇ 2
≡⟨⟩
  3 ≤ᵇ 1
≡⟨⟩
  2 ≤ᵇ 0
≡⟨⟩
  false
∎
```

In the first case, it takes two steps to reduce the first argument to zero, and one more step to compute true, corresponding to the two uses of `s≤s` and the one use of `z≤n` when providing evidence that `2 ≤ 4`. In the second case, it takes two steps to reduce the second argument to zero, and one more step to compute false, corresponding to the two uses of `s≤s` and the one use of `()` when showing there can be no evidence that `4 ≤ 2`.

Relating evidence and computation

We would hope to be able to show these two approaches are related, and indeed we can. First, we define a function that lets us map from the computation world to the evidence world:

```
T : Bool → Set
T true  = ⊤
T false = ⊥
```

Recall that `⊤` is the unit type which contains the single element `tt`, and the `⊥` is the empty type which contains no values. (Also note that `T` is a capital letter t, and distinct from `⊤`.) If `b` is of type `Bool`, then `tt` provides evidence that `T b` holds if `b` is true, while there is no possible evidence that `T b` holds if `b` is false.

Another way to put this is that `T b` is inhabited exactly when `b ≡ true` is inhabited. In the forward direction, we need to do a case analysis on the boolean `b`:

```
T→≡ : ∀ (b : Bool) → T b → b ≡ true
T→≡ true tt = refl
T→≡ false ()
```

If `b` is true then `T b` is inhabited by `tt` and `b ≡ true` is inhabited by `refl`, while if `b` is false then `T b` in uninhabited.

In the reverse direction, there is no need for a case analysis on the boolean `b`:

```
≡→T : ∀ {b : Bool} → b ≡ true → T b
≡→T refl = tt
```

If `b ≡ true` is inhabited by `refl` we know that `b` is `true` and hence `T b` is inhabited by `tt`.

Now we can show that `T (m ≤ᵇ n)` is inhabited exactly when `m ≤ n` is inhabited.

In the forward direction, we consider the three clauses in the definition of `_≤ᵇ_`:

```
≤ᵇ→≤ : ∀ (m n : ℕ) → T (m ≤ᵇ n) → m ≤ n
≤ᵇ→≤ zero n       tt  = z≤n
≤ᵇ→≤ (suc m) zero ()
≤ᵇ→≤ (suc m) (suc n) t = s≤s (≤ᵇ→≤ m n t)
```

In the first clause, we immediately have that `zero ≤ᵇ n` is true, so `T (m ≤ᵇ n)` is evidenced by `tt`, and correspondingly `m ≤ n` is evidenced by `z≤n`. In the middle clause, we immediately have that `suc m ≤ᵇ zero` is false, and hence `T (m ≤ᵇ n)` is empty, so we need not provide evidence that `m ≤ n`, which is just as well since there is no such evidence. In the last clause, we have that `suc m ≤ᵇ suc n` recurses to `m ≤ᵇ n`. We let `t` be the evidence of `T (suc m ≤ᵇ suc n)` if it exists, which, by definition of `_≤ᵇ_`, will also be evidence of `T (m ≤ᵇ n)`. We recursively invoke the function to get evidence that `m ≤ n`, which `s≤s` converts to evidence that `suc m ≤ suc n`.

In the reverse direction, we consider the possible forms of evidence that `m ≤ n`:

```
≤→≤ᵇ : ∀ {m n : ℕ} → m ≤ n → T (m ≤ᵇ n)
≤→≤ᵇ z≤n       = tt
≤→≤ᵇ (s≤s m≤n) = ≤→≤ᵇ m≤n
```

If the evidence is `z≤n` then we immediately have that `zero ≤ᵇ n` is true, so `T (m ≤ᵇ n)` is evidenced by `tt`. If the evidence is `s≤s` applied to `m≤n`, then `suc m ≤ᵇ suc n` reduces to `m ≤ᵇ n`, and we may recursively invoke the function to produce evidence that `T (m ≤ᵇ n)`.

The forward proof has one more clause than the reverse proof, precisely because in the forward proof we need clauses corresponding to the comparison yielding both true and false, while in the reverse proof we only need clauses corresponding to the case where there is evidence that the comparison holds. This is exactly why we tend to prefer the evidence formulation to the computation formulation, because it allows us to do less work: we consider only cases where the relation holds, and can ignore those where it does not.

On the other hand, sometimes the computation formulation may be just what we want. Given a non-obvious relation over large values, it might be handy to have the computer work out the answer for us. Fortunately, rather than choosing between *evidence* and *computation*, there is a way to get the benefits of both.

The best of both worlds

A function that returns a boolean returns exactly a single bit of information: does the relation hold or does it not? Conversely, the evidence approach tells us exactly why the relation holds, but we are responsible for generating the evidence. But it is easy to define a type that combines the benefits of both approaches. It is called `Dec A`, where `Dec` is short for *decidable*:

```
data Dec (A : Set) : Set where
  yes : A → Dec A
  no : ¬ A → Dec A
```

Like booleans, the type has two constructors. A value of type `Dec A` is either of the form `yes x`, where `x` provides evidence that `A` holds, or of the form `no ¬x`, where `¬x` provides evidence that `A` cannot hold (that is, `¬x` is a function which given evidence of `A` yields a contradiction).

For example, we define a function `_≤?_` which given two numbers decides whether one is less than or equal to the other, and provides evidence to justify its conclusion.

First, we introduce two functions useful for constructing evidence that an inequality does not hold:

```
¬s≤z : ∀ {m : ℕ} → ¬ (suc m ≤ zero)
¬s≤z ()

¬s≤s : ∀ {m n : ℕ} → ¬ (m ≤ n) → ¬ (suc m ≤ suc n)
¬s≤s ¬m≤n (s≤s m≤n) = ¬m≤n m≤n
```

The first of these asserts that `¬ (suc m ≤ zero)`, and follows by absurdity, since any evidence of inequality has the form `zero ≤ n` or `suc m ≤ suc n`, neither of which match `suc m ≤ zero`. The second of these takes evidence `¬m≤n` of `¬ (m ≤ n)` and returns a proof of `¬ (suc m ≤ suc n)`. Any evidence of `suc m ≤ suc n` must have the form `s≤s m≤n` where `m≤n` is evidence that `m ≤ n`. Hence, we have a contradiction, evidenced by `¬m≤n m≤n`.

Using these, it is straightforward to decide an inequality:

```
_≤?_ : ∀ (m n : ℕ) → Dec (m ≤ n)
zero ≤? n     = yes z≤n
suc m ≤? zero = no ¬s≤z
suc m ≤? suc n with m ≤? n
... | yes m≤n = yes (s≤s m≤n)
... | no ¬m≤n = no (¬s≤s ¬m≤n)
```

As with `_≤ᵇ_`, the definition has three clauses. In the first clause, it is immediate that `zero ≤ n` holds, and it is evidenced by `z≤n`. In the second clause, it is immediate that `suc m ≤ zero` does not hold, and it is evidenced by `¬s≤z`. In the third clause, to decide whether `suc m ≤ suc n` holds we recursively invoke `m ≤? n`. There are two possibilities. In the `yes` case it returns evidence `m≤n` that `m ≤ n`, and `s≤s m≤n` provides evidence that `suc m ≤ suc n`. In the `no` case it returns evidence `¬m≤n` that `¬ (m ≤ n)`, and `¬s≤s ¬m≤n` provides evidence that `¬ (suc m ≤ suc n)`.

When we wrote `_≤ᵇ_`, we had to write two other functions, `≤ᵇ→≤` and `≤→≤ᵇ`, in order to show that it was correct. In contrast, the definition of `_≤?_` proves itself correct, as attested to by its type. The code of `_≤?_` is far more compact than the combined code of `_≤ᵇ_`, `≤ᵇ→≤`, and `≤→≤ᵇ`. As we will later show, if you really want the latter three, it is easy to derive them from `_≤?_`.

We can use our new function to *compute* the *evidence* that earlier we had to think up on our own:

```
_ : 2 ≤? 4 ≡ yes (s≤s (s≤s z≤n))
_ = refl

_ : 4 ≤? 2 ≡ no (¬s≤s (¬s≤s ¬s≤z))
_ = refl
```

You can check that Agda will indeed compute these values. Typing `C-c C-n` and providing `2 ≤? 4` or `4 ≤? 2` as the requested expression causes Agda to print the values given above.

(A subtlety: if we do not define `¬s≤z` and `¬s≤s` as top-level functions, but instead use inline anonymous functions then Agda may have trouble normalising evidence of negation.)

Exercise `_<?_` (recommended)

Analogous to the function above, define a function to decide strict inequality:

```
postulate
  _<?_ : ∀ (m n : ℕ) → Dec (m < n)
```

```
-- Your code goes here
```

Exercise `_≡ℕ?_` (practice)

Define a function to decide whether two naturals are equal:

```
postulate
  _≡ℕ?_ : ∀ (m n : ℕ) → Dec (m ≡ n)
```

```
-- Your code goes here
```

Decidables from booleans, and booleans from decidables

Curious readers might wonder if we could reuse the definition of `m ≤ᵇ n`, together with the proofs that it is equivalent to `m ≤ n`, to show decidability. Indeed, we can do so as follows:

```
_≤?′_ : ∀ (m n : ℕ) → Dec (m ≤ n)
m ≤?′ n with m ≤ᵇ n | ≤ᵇ→≤ m n | ≤→≤ᵇ {m} {n}
... | true  | p | _  = yes (p tt)
... | false | _ | ¬p = no ¬p
```

If `m ≤ᵇ n` is true then `≤ᵇ→≤` yields a proof that `m ≤ n` holds, while if it is false then `≤→≤ᵇ` takes a proof that `m ≤ n` holds into a contradiction.

The triple binding of the `with` clause in this proof is essential. If instead we wrote:

```
_≤?″_ : ∀ (m n : ℕ) → Dec (m ≤ n)
m ≤?″ n with m ≤ᵇ n
... | true   =  yes (≤ᵇ→≤ m n tt)
... | false  =  no (≤→≤ᵇ {m} {n})
```

then Agda would make two complaints, one for each clause:

```
T !=< (T (m ≤ᵇ n)) of type Set
when checking that the expression tt has type T (m ≤ᵇ n)

T (m ≤ᵇ n) !=< ⊥ of type Set
when checking that the expression ≤→≤ᵇ {m} {n} has type ¬ m ≤ n
```

Putting the expressions into the `with` clause permits Agda to exploit the fact that `T (m ≤ᵇ n)` is `T` when `m ≤ᵇ n` is true, and that `T (m ≤ᵇ n)` is `⊥` when `m ≤ᵇ n` is false.

However, overall it is simpler to just define `_≤?_` directly, as in the previous section. If one really wants `_≤ᵇ_`, then it and its properties are easily derived from `_≤?_`, as we will now show.

Erasure takes a decidable value to a boolean:

```
⌊_⌋ : ∀ {A : Set} → Dec A → Bool
⌊ yes x ⌋ = true
⌊ no ¬x ⌋ = false
```

Using erasure, we can easily derive `_≤ᵇ_` from `_≤?_`:

```
_≤ᵇ′_ : ℕ → ℕ → Bool
m ≤ᵇ′ n = ⌊ m ≤? n ⌋
```

Further, if `D` is a value of type `Dec A`, then `T ⌊ D ⌋` is inhabited exactly when `A` is inhabited:

```
toWitness : ∀ {A : Set} {D : Dec A} → T ⌊ D ⌋ → A
toWitness {A} {yes x} tt = x
toWitness {A} {no ¬x} ()

fromWitness : ∀ {A : Set} {D : Dec A} → A → T ⌊ D ⌋
fromWitness {A} {yes x} _ = tt
fromWitness {A} {no ¬x} x = ¬x x
```

Using these, we can easily derive that `T (m ≤ᵇ′ n)` is inhabited exactly when `m ≤ n` is inhabited:

```
≤ᵇ′→≤ : ∀ {m n : ℕ} → T (m ≤ᵇ′ n) → m ≤ n
≤ᵇ′→≤ = toWitness

≤→≤ᵇ′ : ∀ {m n : ℕ} → m ≤ n → T (m ≤ᵇ′ n)
≤→≤ᵇ′ = fromWitness
```

In summary, it is usually best to eschew booleans and rely on decidables. If you need booleans, they and their properties are easily derived from the corresponding decidables.

Logical connectives

Most readers will be familiar with the logical connectives for booleans. Each of these extends to decidables.

The conjunction of two booleans is true if both are true, and false if either is false:

```
infixr 6 _∧_

_∧_ : Bool → Bool → Bool
true ∧ true = true
false ∧ _    = false
_     ∧ false = false
```

In Emacs, the left-hand side of the third equation displays in grey, indicating that the order of the equations determines which of the second or the third can match. However, regardless of which matches the answer is the same.

Correspondingly, given two decidable propositions, we can decide their conjunction:

```
infixr 6 _×-dec_

_×-dec_ : ∀ {A B : Set} → Dec A → Dec B → Dec (A × B)
yes x ×-dec yes y = yes ⟨ x , y ⟩
no ¬x ×-dec _ = no λ{ ⟨ x , y ⟩ → ¬x x }
```

```
_×-dec no ¬y = no λ{ ⟨ x , y ⟩ → ¬y y }
```

The conjunction of two propositions holds if they both hold, and its negation holds if the negation of either holds. If both hold, then we pair the evidence for each to yield evidence of the conjunct. If the negation of either holds, assuming the conjunct will lead to a contradiction.

Again in Emacs, the left-hand side of the third equation displays in grey, indicating that the order of the equations determines which of the second or the third can match. This time the answer is different depending on which matches; if both conjuncts fail to hold we pick the first to yield the contradiction, but it would be equally valid to pick the second.

The disjunction of two booleans is true if either is true, and false if both are false:

```
infixr 5 _∨_

_∨_ : Bool → Bool → Bool
true ∨ _     = true
_    ∨ true  = true
false ∨ false = false
```

In Emacs, the left-hand side of the second equation displays in grey, indicating that the order of the equations determines which of the first or the second can match. However, regardless of which matches the answer is the same.

Correspondingly, given two decidable propositions, we can decide their disjunction:

```
infixr 5 _⊎-dec_

_⊎-dec_ : ∀ {A B : Set} → Dec A → Dec B → Dec (A ⊎ B)
yes x ⊎-dec _ = yes (inj₁ x)
_ ⊎-dec yes y = yes (inj₂ y)
no ¬x ⊎-dec no ¬y = no λ{ (inj₁ x) → ¬x x ; (inj₂ y) → ¬y y }
```

The disjunction of two propositions holds if either holds, and its negation holds if the negation of both hold. If either holds, we inject the evidence to yield evidence of the disjunct. If the negation of both hold, assuming either disjunct will lead to a contradiction.

Again in Emacs, the left-hand side of the second equation displays in grey, indicating that the order of the equations determines which of the first or the second can match. This time the answer is different depending on which matches; if both disjuncts hold we pick the first, but it would be equally valid to pick the second.

The negation of a boolean is false if its argument is true, and vice versa:

```
not : Bool → Bool
not true = false
not false = true
```

Correspondingly, given a decidable proposition, we can decide its negation:

```
¬? : ∀ {A : Set} → Dec A → Dec (¬ A)
¬? (yes x) = no (¬¬-intro x)
¬? (no ¬x) = yes ¬x
```

We simply swap yes and no. In the first equation, the right-hand side asserts that the negation of `¬ A` holds, in other words, that `¬ ¬ A` holds, which is an easy consequence of the fact that `A` holds.

There is also a slightly less familiar connective, corresponding to implication:

```
_⊃_ : Bool → Bool → Bool
_     ⊃ true  = true
false ⊃ _     = true
true  ⊃ false = false
```

One boolean implies another if whenever the first is true then the second is true. Hence, the implication of two booleans is true if the second is true or the first is false, and false if the first is true and the second is false. In Emacs, the left-hand side of the second equation displays in grey, indicating that the order of the equations determines which of the first or the second can match. However, regardless of which matches the answer is the same.

Correspondingly, given two decidable propositions, we can decide if the first implies the second:

```
_→-dec_ : ∀ {A B : Set} → Dec A → Dec B → Dec (A → B)
_ →-dec yes y      = yes (λ _ → y)
no ¬x →-dec _      = yes (λ x → ⊥-elim (¬x x))
yes x →-dec no ¬y = no (λ f → ¬y (f x))
```

The implication holds if either the second holds or the negation of the first holds, and its negation holds if the first holds and the negation of the second holds. Evidence for the implication is a function from evidence of the first to evidence of the second. If the second holds, the function returns the evidence for it. If the negation of the first holds, the function takes the evidence of the first and derives a contradiction. If the first holds and the negation of the second holds, given evidence of the implication we must derive a contradiction; we apply the evidence of the implication `f` to the evidence of the first `x`, yielding a contradiction with the evidence `¬y` of the negation of the second.

Again in Emacs, the left-hand side of the second equation displays in grey, indicating that the order of the equations determines which of the first or the second can match. This time the answer is different depending on which matches; but either is equally valid.

Exercise `erasure` (practice)

Show that erasure relates corresponding boolean and decidable operations:

```
postulate
  ∧-× : ∀ {A B : Set} (x : Dec A) (y : Dec B) → ⌊ x ⌋ ∧ ⌊ y ⌋ ≡ ⌊ x ×-dec y ⌋
  ∨-⊎ : ∀ {A B : Set} (x : Dec A) (y : Dec B) → ⌊ x ⌋ ∨ ⌊ y ⌋ ≡ ⌊ x ⊎-dec y ⌋
  not-¬ : ∀ {A : Set} (x : Dec A) → not ⌊ x ⌋ ≡ ⌊ ¬? x ⌋
```

Exercise `iff-erasure` (recommended)

Give analogues of the `_⇔_` operation from Chapter Isomorphism, operation on booleans and decidables, and also show the corresponding erasure:

```
postulate
  _iff_ : Bool → Bool → Bool
  _⇔-dec_ : ∀ {A B : Set} → Dec A → Dec B → Dec (A ⇔ B)
  iff-⇔ : ∀ {A B : Set} (x : Dec A) (y : Dec B) → ⌊ x ⌋ iff ⌊ y ⌋ ≡ ⌊ x ⇔-dec y ⌋
```

```
-- Your code goes here
```

Proof by reflection

Let's revisit our definition of monus from Chapter Naturals. If we subtract a larger number from a smaller number, we take the result to be zero. We had to do something, after all. What could we have done differently? We could have defined a *guarded* version of minus, a function which subtracts `n` from `m` only if `n ≤ m`:

```
minus : (m n : ℕ) (n≤m : n ≤ m) → ℕ
minus m zero _ = m
minus (suc m) (suc n) (s≤s n≤m) = minus m n n≤m
```

Unfortunately, it is painful to use, since we have to explicitly provide the proof that `n ≤ m`:

```
_ : minus 5 3 (s≤s (s≤s (s≤s z≤n))) ≡ 2
_ = refl
```

We cannot solve this problem in general, but in the scenario above, we happen to know the two numbers *statically*. In that case, we can use a technique called *proof by reflection*. Essentially, we can ask Agda to run the decidable equality `n ≤? m` while type checking, and make sure that `n ≤ m`!

We do this by using a feature of implicits. Agda will fill in an implicit of a record type if it can fill in all its fields. So Agda will *always* manage to fill in an implicit of an *empty* record type, since there aren't any fields after all. This is why `⊤` is defined as an empty record.

The trick is to have an implicit argument of the type `T ⌊ n ≤? m ⌋`. Let's go through what this means step-by-step. First, we run the decision procedure, `n ≤? m`. This provides us with evidence whether `n ≤ m` holds or not. We erase the evidence to a boolean. Finally, we apply `T`. Recall that `T` maps booleans into the world of evidence: `true` becomes the unit type `⊤`, and `false` becomes the empty type `⊥`. Operationally, an implicit argument of this type works as a guard.

- If `n ≤ m` holds, the type of the implicit value reduces to `⊤`. Agda then happily provides the implicit value.
- Otherwise, the type reduces to `⊥`, which Agda has no chance of providing, so it will throw an error. For instance, if we call `3 - 5` we get `_n≤m_254 : ⊥`.

We obtain the witness for `n ≤ m` using `toWitness`, which we defined earlier:

```
_-_ : (m n : ℕ) {n≤m : T ⌊ n ≤? m ⌋} → ℕ
_-_ m n {n≤m} = minus m n (toWitness n≤m)
```

We can safely use `_-_` as long as we statically know the two numbers:

```
_ : 5 - 3 ≡ 2
_ = refl
```

It turns out that this idiom is very common. The standard library defines a synonym for `T ⌊ ? ⌋` called `True`:

```
True : ∀ {Q} → Dec Q → Set
True Q = T ⌊ Q ⌋
```

Exercise `False`

Give analogues of `True`, `toWitness`, and `fromWitness` which work with *negated* properties. Call these `False`, `toWitnessFalse`, and `fromWitnessFalse`.

Standard Library

```
import Data.Bool.Base using (Bool; true; false; T; _∧_; _∨_; not)
import Data.Nat using (_≤?_)
import Relation.Nullary using (Dec; yes; no)
import Relation.Nullary.Decidable using (⌊_⌋; True; toWitness; fromWitness)
import Relation.Nullary.Negation using (¬?)
import Relation.Nullary.Product using (_×-dec_)
import Relation.Nullary.Sum using (_⊎-dec_)
import Relation.Binary using (Decidable)
```

Unicode

```
∧  U+2227  LOGICAL AND (\and, \wedge)
∨  U+2228  LOGICAL OR (\or, \vee)
⊃  U+2283  SUPERSET OF (\sup)
ᵇ  U+1D47  MODIFIER LETTER SMALL B  (\^b)
⌊  U+230A  LEFT FLOOR (\clL)
⌋  U+230B  RIGHT FLOOR (\clR)
```

Chapter 10

Lists: Lists and higher-order functions

```
module plfa.part1.Lists where
```

This chapter discusses the list data type. It gives further examples of many of the techniques we have developed so far, and provides examples of polymorphic types and higher-order functions.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; trans; cong)
open Eq.≡-Reasoning
open import Data.Bool using (Bool; true; false; T; _∧_; _∨_; not)
open import Data.Nat using (ℕ; zero; suc; _+_; _*_; _∸_; _≤_; s≤s; z≤n)
open import Data.Nat.Properties using
  (+-assoc; +-identityˡ; +-identityʳ; *-assoc; *-identityˡ; *-identityʳ)
open import Relation.Nullary using (¬_; Dec; yes; no)
open import Data.Product using (_×_; ∃; ∃-syntax) renaming (_,_ to ⟨_,_⟩)
open import Function using (_∘_)
open import Level using (Level)
open import plfa.part1.Isomorphism using (_≃_; _⇔_)
```

Lists

Lists are defined in Agda as follows:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

infixr 5 _::_
```

Let's unpack this definition. If `A` is a set, then `List A` is a set. The next two lines tell us that `[]` (pronounced *nil*) is a list of type `A` (often called the *empty* list), and that `_::_` (pronounced *cons*,

short for *constructor*) takes a value of type `A` and a value of type `List A` and returns a value of type `List A`. Operator `_::_` has precedence level 5 and associates to the right.

For example,

```
_ : List ℕ
_ = 0 :: 1 :: 2 :: []
```

denotes the list of the first three natural numbers. Since `_::_` associates to the right, the term parses as `0 :: (1 :: (2 :: []))`. Here `0` is the first element of the list, called the *head*, and `1 :: (2 :: [])` is a list of the remaining elements, called the *tail*. A list is a strange beast: it has a head and a tail, nothing in between, and the tail is itself another list!

As we've seen, parameterised types can be translated to indexed types. The definition above is equivalent to the following:

```
data List′ : Set → Set where
  []′ : ∀ {A : Set} → List′ A
  _::′_ : ∀ {A : Set} → A → List′ A → List′ A
```

Each constructor takes the parameter as an implicit argument. Thus, our example list could also be written:

```
_ : List ℕ
_ = _::_ {ℕ} 0 (_::_ {ℕ} 1 (_::_ {ℕ} 2 ([] {ℕ})))
```

where here we have provided the implicit parameters explicitly.

Including the pragma:

```
{-# BUILTIN LIST List #-}
```

tells Agda that the type `List` corresponds to the Haskell type list, and the constructors `[]` and `_::_` correspond to nil and cons respectively, allowing a more efficient representation of lists.

List syntax

We can write lists more conveniently by introducing the following definitions:

```
pattern [_] z = z :: []
pattern [_,_] y z = y :: z :: []
pattern [_,_,_] x y z = x :: y :: z :: []
pattern [_,_,_,_] w x y z = w :: x :: y :: z :: []
pattern [_,_,_,_,_] v w x y z = v :: w :: x :: y :: z :: []
pattern [_,_,_,_,_,_] u v w x y z = u :: v :: w :: x :: y :: z :: []
```

This is our first use of pattern declarations. For instance, the third line tells us that `[ x , y , z ]` is equivalent to `x :: y :: z :: []`, and permits the former to appear either in a pattern on the left-hand side of an equation, or a term on the right-hand side of an equation.

Append

Our first function on lists is written `_++_` and pronounced *append*:

```
infixr 5 _++_

_++_ : ∀ {A : Set} → List A → List A → List A
[] ++ ys       = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

The type `A` is an implicit argument to append, making it a *polymorphic* function (one that can be used at many types). A list appended to the empty list yields the list itself. A list appended to a non-empty list yields a list with the head the same as the head of the non-empty list, and a tail the same as the other list appended to tail of the non-empty list.

Here is an example, showing how to compute the result of appending two lists:

```
_ : [ 0 , 1 , 2 ] ++ [ 3 , 4 ] ≡ [ 0 , 1 , 2 , 3 , 4 ]
_ =
  begin
    0 :: 1 :: 2 :: [] ++ 3 :: 4 :: []
  ≡⟨⟩
    0 :: (1 :: 2 :: [] ++ 3 :: 4 :: [])
  ≡⟨⟩
    0 :: 1 :: (2 :: [] ++ 3 :: 4 :: [])
  ≡⟨⟩
    0 :: 1 :: 2 :: ([] ++ 3 :: 4 :: [])
  ≡⟨⟩
    0 :: 1 :: 2 :: 3 :: 4 :: []
  ∎
```

Appending two lists requires time linear in the number of elements in the first list.

Reasoning about append

We can reason about lists in much the same way that we reason about numbers. Here is the proof that append is associative:

```
++-assoc : ∀ {A : Set} (xs ys zs : List A)
  → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc [] ys zs =
  begin
    ([] ++ ys) ++ zs
  ≡⟨⟩
    ys ++ zs
  ≡⟨⟩
    [] ++ (ys ++ zs)
  ∎
++-assoc (x :: xs) ys zs =
  begin
    (x :: xs ++ ys) ++ zs
  ≡⟨⟩
    x :: (xs ++ ys) ++ zs
  ≡⟨⟩
    x :: ((xs ++ ys) ++ zs)
```

```
  ≡⟨ cong (x ::_) (++-assoc xs ys zs) ⟩
    x :: (xs ++ (ys ++ zs))
  ≡⟨⟩
    x :: xs ++ (ys ++ zs)
  ∎
```

The proof is by induction on the first argument. The base case instantiates to `[]`, and follows by straightforward computation. The inductive case instantiates to `x :: xs`, and follows by straightforward computation combined with the inductive hypothesis. As usual, the inductive hypothesis is indicated by a recursive invocation of the proof, in this case `++-assoc xs ys zs`.

Recall that Agda supports sections. Applying `cong (x ::_)` promotes the inductive hypothesis:

```
(xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
```

to the equality:

```
x :: ((xs ++ ys) ++ zs) ≡ x :: (xs ++ (ys ++ zs))
```

which is needed in the proof.

It is also easy to show that `[]` is a left and right identity for `_++_`. That it is a left identity is immediate from the definition:

```
++-identityˡ : ∀ {A : Set} (xs : List A) → [] ++ xs ≡ xs
++-identityˡ xs =
  begin
    [] ++ xs
  ≡⟨⟩
    xs
  ∎
```

That it is a right identity follows by simple induction:

```
++-identityʳ : ∀ {A : Set} (xs : List A) → xs ++ [] ≡ xs
++-identityʳ [] =
  begin
    [] ++ []
  ≡⟨⟩
    []
  ∎
++-identityʳ (x :: xs) =
  begin
    (x :: xs) ++ []
  ≡⟨⟩
    x :: (xs ++ [])
  ≡⟨ cong (x ::_) (++-identityʳ xs) ⟩
    x :: xs
  ∎
```

As we will see later, these three properties establish that `_++_` and `[]` form a *monoid* over lists.

Length

Our next function finds the length of a list:

```
length : ∀ {A : Set} → List A → ℕ
length []       = zero
length (x :: xs) = suc (length xs)
```

Again, it takes an implicit parameter `A`. The length of the empty list is zero. The length of a non-empty list is one greater than the length of the tail of the list.

Here is an example showing how to compute the length of a list:

```
_ : length [ 0 , 1 , 2 ] ≡ 3
_ =
  begin
    length (0 :: 1 :: 2 :: [])
  ≡⟨⟩
    suc (length (1 :: 2 :: []))
  ≡⟨⟩
    suc (suc (length (2 :: [])))
  ≡⟨⟩
    suc (suc (suc (length {ℕ} [])))
  ≡⟨⟩
    suc (suc (suc zero))
  ∎
```

Computing the length of a list requires time linear in the number of elements in the list.

In the second-to-last line, we cannot write simply `length []` but must instead write `length {ℕ} []`. Since `[]` has no elements, Agda has insufficient information to infer the implicit parameter.

Reasoning about length

The length of one list appended to another is the sum of the lengths of the lists:

```
length-++ : ∀ {A : Set} (xs ys : List A)
  → length (xs ++ ys) ≡ length xs + length ys
length-++ {A} [] ys =
  begin
    length ([] ++ ys)
  ≡⟨⟩
    length ys
  ≡⟨⟩
    length {A} [] + length ys
  ∎
length-++ (x :: xs) ys =
  begin
    length ((x :: xs) ++ ys)
  ≡⟨⟩
    suc (length (xs ++ ys))
  ≡⟨ cong suc (length-++ xs ys) ⟩
    suc (length xs + length ys)
  ≡⟨⟩
    length (x :: xs) + length ys
  ∎
```

The proof is by induction on the first argument. The base case instantiates to `[]`, and follows

by straightforward computation. As before, Agda cannot infer the implicit type parameter to `length`, and it must be given explicitly. The inductive case instantiates to `x :: xs`, and follows by straightforward computation combined with the inductive hypothesis. As usual, the inductive hypothesis is indicated by a recursive invocation of the proof, in this case `length-++ xs ys`, and it is promoted by the congruence `cong suc`.

Reverse

Using append, it is easy to formulate a function to reverse a list:

```
reverse : ∀ {A : Set} → List A → List A
reverse []       = []
reverse (x :: xs) = reverse xs ++ [ x ]
```

The reverse of the empty list is the empty list. The reverse of a non-empty list is the reverse of its tail appended to a unit list containing its head.

Here is an example showing how to reverse a list:

```
_ : reverse [ 0 , 1 , 2 ] ≡ [ 2 , 1 , 0 ]
_ =
  begin
    reverse (0 :: 1 :: 2 :: [])
  ≡⟨⟩
    reverse (1 :: 2 :: []) ++ [ 0 ]
  ≡⟨⟩
    (reverse (2 :: []) ++ [ 1 ]) ++ [ 0 ]
  ≡⟨⟩
    ((reverse [] ++ [ 2 ]) ++ [ 1 ]) ++ [ 0 ]
  ≡⟨⟩
    (([] ++ [ 2 ]) ++ [ 1 ]) ++ [ 0 ]
  ≡⟨⟩
    (([] ++ 2 :: []) ++ 1 :: []) ++ 0 :: []
  ≡⟨⟩
    (2 :: [] ++ 1 :: []) ++ 0 :: []
  ≡⟨⟩
    2 :: ([] ++ 1 :: []) ++ 0 :: []
  ≡⟨⟩
    (2 :: 1 :: []) ++ 0 :: []
  ≡⟨⟩
    2 :: (1 :: [] ++ 0 :: [])
  ≡⟨⟩
    2 :: 1 :: ([] ++ 0 :: [])
  ≡⟨⟩
    2 :: 1 :: 0 :: []
  ≡⟨⟩
    [ 2 , 1 , 0 ]
  ∎
```

Reversing a list in this way takes time *quadratic* in the length of the list. This is because reverse ends up appending lists of lengths `1`, `2`, up to `n - 1`, where `n` is the length of the list being reversed, append takes time linear in the length of the first list, and the sum of the numbers up to `n - 1` is `n * (n - 1) / 2`. (We will validate that last fact in an exercise later in this chapter.)

Exercise `reverse-++-distrib` **(recommended)**

Show that the reverse of one list appended to another is the reverse of the second appended to the reverse of the first:

```
reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
```

Exercise `reverse-involutive` **(recommended)**

A function is an *involution* if when applied twice it acts as the identity function. Show that reverse is an involution:

```
reverse (reverse xs) ≡ xs
```

Faster reverse

The definition above, while easy to reason about, is less efficient than one might expect since it takes time quadratic in the length of the list. The idea is that we generalise reverse to take an additional argument:

```
shunt : ∀ {A : Set} → List A → List A → List A
shunt [] ys       = ys
shunt (x :: xs) ys = shunt xs (x :: ys)
```

The definition is by recursion on the first argument. The second argument actually becomes *larger*, but this is not a problem because the argument on which we recurse becomes *smaller*.

Shunt is related to reverse as follows:

```
shunt-reverse : ∀ {A : Set} (xs ys : List A)
  → shunt xs ys ≡ reverse xs ++ ys
shunt-reverse [] ys =
  begin
    shunt [] ys
  ≡⟨⟩
    ys
  ≡⟨⟩
    reverse [] ++ ys
  ∎
shunt-reverse (x :: xs) ys =
  begin
    shunt (x :: xs) ys
  ≡⟨⟩
    shunt xs (x :: ys)
  ≡⟨ shunt-reverse xs (x :: ys) ⟩
    reverse xs ++ (x :: ys)
  ≡⟨⟩
    reverse xs ++ ([ x ] ++ ys)
  ≡⟨ sym (++-assoc (reverse xs) [ x ] ys) ⟩
    (reverse xs ++ [ x ]) ++ ys
  ≡⟨⟩
    reverse (x :: xs) ++ ys
```

```
  ∎
```

The proof is by induction on the first argument. The base case instantiates to `[]`, and follows by straightforward computation. The inductive case instantiates to `x :: xs` and follows by the inductive hypothesis and associativity of append. When we invoke the inductive hypothesis, the second argument actually becomes *larger*, but this is not a problem because the argument on which we induct becomes *smaller*.

Generalising on an auxiliary argument, which becomes larger as the argument on which we recurse or induct becomes smaller, is a common trick. It belongs in your quiver of arrows, ready to slay the right problem.

Having defined shunt be generalisation, it is now easy to respecialise to give a more efficient definition of reverse:

```
reverse′ : ∀ {A : Set} → List A → List A
reverse′ xs = shunt xs []
```

Given our previous lemma, it is straightforward to show the two definitions equivalent:

```
reverses : ∀ {A : Set} (xs : List A)
  → reverse′ xs ≡ reverse xs
reverses xs =
  begin
    reverse′ xs
  ≡⟨⟩
    shunt xs []
  ≡⟨ shunt-reverse xs [] ⟩
    reverse xs ++ []
  ≡⟨ ++-identityʳ (reverse xs) ⟩
    reverse xs
  ∎
```

Here is an example showing fast reverse of the list `[ 0 , 1 , 2 ]`:

```
_ : reverse′ [ 0 , 1 , 2 ] ≡ [ 2 , 1 , 0 ]
_ =
  begin
    reverse′ (0 :: 1 :: 2 :: [])
  ≡⟨⟩
    shunt (0 :: 1 :: 2 :: []) []
  ≡⟨⟩
    shunt (1 :: 2 :: []) (0 :: [])
  ≡⟨⟩
    shunt (2 :: []) (1 :: 0 :: [])
  ≡⟨⟩
    shunt [] (2 :: 1 :: 0 :: [])
  ≡⟨⟩
    2 :: 1 :: 0 :: []
  ∎
```

Now the time to reverse a list is linear in the length of the list.

Map

Map applies a function to every element of a list to generate a corresponding list. Map is an example of a *higher-order function*, one which takes a function as an argument or returns a function as a result:

```
map : ∀ {A B : Set} → (A → B) → List A → List B
map f []       = []
map f (x :: xs) = f x :: map f xs
```

Map of the empty list is the empty list. Map of a non-empty list yields a list with head the same as the function applied to the head of the given list, and tail the same as map of the function applied to the tail of the given list.

Here is an example showing how to use map to increment every element of a list:

```
_ : map suc [ 0 , 1 , 2 ] ≡ [ 1 , 2 , 3 ]
_ =
  begin
    map suc (0 :: 1 :: 2 :: [])
  ≡⟨⟩
    suc 0 :: map suc (1 :: 2 :: [])
  ≡⟨⟩
    suc 0 :: suc 1 :: map suc (2 :: [])
  ≡⟨⟩
    suc 0 :: suc 1 :: suc 2 :: map suc []
  ≡⟨⟩
    suc 0 :: suc 1 :: suc 2 :: []
  ≡⟨⟩
    1 :: 2 :: 3 :: []
  ∎
```

Map requires time linear in the length of the list.

It is often convenient to exploit currying by applying map to a function to yield a new function, and at a later point applying the resulting function:

```
sucs : List ℕ → List ℕ
sucs = map suc

_ : sucs [ 0 , 1 , 2 ] ≡ [ 1 , 2 , 3 ]
_ =
  begin
    sucs [ 0 , 1 , 2 ]
  ≡⟨⟩
    map suc [ 0 , 1 , 2 ]
  ≡⟨⟩
    [ 1 , 2 , 3 ]
  ∎
```

Any type that is parameterised on another type, such as lists, has a corresponding map, which accepts a function and returns a function from the type parameterised on the domain of the function to the type parameterised on the range of the function. Further, a type that is parameterised on *n* types will have a map that is parameterised on *n* functions.

Exercise `map-compose` (practice)

Prove that the map of a composition is equal to the composition of two maps:

```
map (g ∘ f) ≡ map g ∘ map f
```

The last step of the proof requires extensionality.

```
-- Your code goes here
```

Exercise `map-++-distribute` (practice)

Prove the following relationship between map and append:

```
map f (xs ++ ys) ≡ map f xs ++ map f ys
```

```
-- Your code goes here
```

Exercise `map-Tree` (practice)

Define a type of trees with leaves of type `A` and internal nodes of type `B`:

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  node : Tree A B → B → Tree A B → Tree A B
```

Define a suitable map operator over trees:

```
map-Tree : ∀ {A B C D : Set} → (A → C) → (B → D) → Tree A B → Tree C D
```

```
-- Your code goes here
```

Fold

Fold takes an operator and a value, and uses the operator to combine each of the elements of the list, taking the given value as the result for the empty list:

```
foldr : ∀ {A B : Set} → (A → B → B) → B → List A → B
foldr _⊗_ e []       = e
foldr _⊗_ e (x :: xs) = x ⊗ foldr _⊗_ e xs
```

Fold of the empty list is the given value. Fold of a non-empty list uses the operator to combine the head of the list and the fold of the tail of the list.

Here is an example showing how to use fold to find the sum of a list:

```
_ : foldr _+_ 0 [ 1 , 2 , 3 , 4 ] ≡ 10
_ =
  begin
    foldr _+_ 0 (1 :: 2 :: 3 :: 4 :: [])
  ≡⟨⟩
    1 + foldr _+_ 0 (2 :: 3 :: 4 :: [])
  ≡⟨⟩
    1 + (2 + foldr _+_ 0 (3 :: 4 :: []))
  ≡⟨⟩
    1 + (2 + (3 + foldr _+_ 0 (4 :: [])))
  ≡⟨⟩
    1 + (2 + (3 + (4 + foldr _+_ 0 [])))
  ≡⟨⟩
    1 + (2 + (3 + (4 + 0)))
  ∎
```

Here we have an instance of `foldr` where `A` and `B` are both `ℕ`. Fold requires time linear in the length of the list.

It is often convenient to exploit currying by applying fold to an operator and a value to yield a new function, and at a later point applying the resulting function:

```
sum : List ℕ → ℕ
sum = foldr _+_ 0

_ : sum [ 1 , 2 , 3 , 4 ] ≡ 10
_ =
  begin
    sum [ 1 , 2 , 3 , 4 ]
  ≡⟨⟩
    foldr _+_ 0 [ 1 , 2 , 3 , 4 ]
  ≡⟨⟩
    10
  ∎
```

Just as the list type has two constructors, `[]` and `_::_`, so the fold function takes two arguments, `e` and `_⊗_` (in addition to the list argument). In general, a data type with *n* constructors will have a corresponding fold function that takes *n* arguments.

As another example, observe that

```
foldr _::_ [] xs ≡ xs
```

Here, if `xs` is of type `List A`, then we see we have an instance of `foldr` where `A` is `A` and `B` is `List A`. It follows that

```
xs ++ ys ≡ foldr _::_ ys xs
```

Demonstrating both these equations is left as an exercise.

Exercise `product` (recommended)

Use fold to define a function to find the product of a list of numbers. For example:

```
product [ 1 , 2 , 3 , 4 ] ≡ 24
```

```
-- Your code goes here
```

Exercise `foldr-++` (recommended)

Show that fold and append are related as follows:

```
foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ (foldr _⊗_ e ys) xs
```

```
-- Your code goes here
```

Exercise `foldr-∷` (practice)

Show

```
foldr _∷_ [] xs ≡ xs
```

Show as a consequence of `foldr-++` above that

```
xs ++ ys ≡ foldr _∷_ ys xs
```

```
-- Your code goes here
```

Exercise `map-is-foldr` (practice)

Show that map can be defined using fold:

```
map f ≡ foldr (λ x xs → f x ∷ xs) []
```

The proof requires extensionality.

```
-- Your code goes here
```

Exercise `fold-Tree` (practice)

Define a suitable fold function for the type of trees given earlier:

```
fold-Tree : ∀ {A B C : Set} → (A → C) → (C → B → C → C) → Tree A B → C
```

```
-- Your code goes here
```

Exercise `map-is-fold-Tree` (practice)

Demonstrate an analogue of `map-is-foldr` for the type of trees.

```
-- Your code goes here
```

Exercise `sum-downFrom` (stretch)

Define a function that counts down as follows:

```
downFrom : ℕ → List ℕ
downFrom zero    = []
downFrom (suc n) = n :: downFrom n
```

For example:

```
_ : downFrom 3 ≡ [ 2 , 1 , 0 ]
_ = refl
```

Prove that the sum of the numbers `(n - 1) + ⋯ + 0` is equal to `n * (n ∸ 1) / 2`:

```
sum (downFrom n) * 2 ≡ n * (n ∸ 1)
```

Monoids

Typically when we use a fold the operator is associative and the value is a left and right identity for the operator, meaning that the operator and the value form a *monoid*.

We can define a monoid as a suitable record type:

```
record IsMonoid {A : Set} (_⊗_ : A → A → A) (e : A) : Set where
  field
    assoc : ∀ (x y z : A) → (x ⊗ y) ⊗ z ≡ x ⊗ (y ⊗ z)
    identityˡ : ∀ (x : A) → e ⊗ x ≡ x
    identityʳ : ∀ (x : A) → x ⊗ e ≡ x

open IsMonoid
```

As examples, sum and zero, multiplication and one, and append and the empty list, are all examples of monoids:

```
+-monoid : IsMonoid _+_ 0
+-monoid =
  record
    { assoc = +-assoc
    ; identityˡ = +-identityˡ
    ; identityʳ = +-identityʳ
    }

*-monoid : IsMonoid _*_ 1
*-monoid =
```

```
  record
    { assoc = *-assoc
    ; identityˡ = *-identityˡ
    ; identityʳ = *-identityʳ
    }

++-monoid : ∀ {A : Set} → IsMonoid {List A} _++_ []
++-monoid =
  record
    { assoc = ++-assoc
    ; identityˡ = ++-identityˡ
    ; identityʳ = ++-identityʳ
    }
```

If `_⊗_` and `e` form a monoid, then we can re-express fold on the same operator and an arbitrary value:

```
foldr-monoid : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) → IsMonoid _⊗_ e →
  ∀ (xs : List A) (y : A) → foldr _⊗_ y xs ≡ foldr _⊗_ e xs ⊗ y
foldr-monoid _⊗_ e ⊗-monoid [] y =
  begin
    foldr _⊗_ y []
  ≡⟨⟩
    y
  ≡⟨ sym (identityˡ ⊗-monoid y) ⟩
    (e ⊗ y)
  ≡⟨⟩
    foldr _⊗_ e [] ⊗ y
  ∎
foldr-monoid _⊗_ e ⊗-monoid (x :: xs) y =
  begin
    foldr _⊗_ y (x :: xs)
  ≡⟨⟩
    x ⊗ (foldr _⊗_ y xs)
  ≡⟨ cong (x ⊗_) (foldr-monoid _⊗_ e ⊗-monoid xs y) ⟩
    x ⊗ (foldr _⊗_ e xs ⊗ y)
  ≡⟨ sym (assoc ⊗-monoid x (foldr _⊗_ e xs) y) ⟩
    (x ⊗ foldr _⊗_ e xs) ⊗ y
  ≡⟨⟩
    foldr _⊗_ e (x :: xs) ⊗ y
  ∎
```

In a previous exercise we showed the following.

```
postulate
  foldr-++ : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) (xs ys : List A) →
    foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ (foldr _⊗_ e ys) xs
```

As a consequence, using a previous exercise, we have the following:

```
foldr-monoid-++ : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) → IsMonoid _⊗_ e →
  ∀ (xs ys : List A) → foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ e xs ⊗ foldr _⊗_ e ys
foldr-monoid-++ _⊗_ e monoid-⊗ xs ys =
  begin
    foldr _⊗_ e (xs ++ ys)
  ≡⟨ foldr-++ _⊗_ e xs ys ⟩
    foldr _⊗_ (foldr _⊗_ e ys) xs
  ≡⟨ foldr-monoid _⊗_ e monoid-⊗ xs (foldr _⊗_ e ys) ⟩
```

```
  foldr _⊗_ e xs ⊗ foldr _⊗_ e ys
```

Exercise `foldl` (practice)

Define a function `foldl` which is analogous to `foldr`, but where operations associate to the left rather than the right. For example:

```
foldr _⊗_ e [ x , y , z ]  =  x ⊗ (y ⊗ (z ⊗ e))
foldl _⊗_ e [ x , y , z ]  =  ((e ⊗ x) ⊗ y) ⊗ z
```

```
-- Your code goes here
```

Exercise `foldr-monoid-foldl` (practice)

Show that if `_⊗_` and `e` form a monoid, then `foldr _⊗_ e` and `foldl _⊗_ e` always compute the same result.

```
-- Your code goes here
```

All

We can also define predicates over lists. Two of the most important are `All` and `Any`.

Predicate `All P` holds if predicate `P` is satisfied by every element of a list:

```
data All {A : Set} (P : A → Set) : List A → Set where
  [] : All P []
  _::_ : ∀ {x : A} {xs : List A} → P x → All P xs → All P (x :: xs)
```

The type has two constructors, reusing the names of the same constructors for lists. The first asserts that `P` holds for every element of the empty list. The second asserts that if `P` holds of the head of a list and for every element of the tail of a list, then `P` holds for every element of the list. Agda uses types to disambiguate whether the constructor is building a list or evidence that `All P` holds.

For example, `All (_≤ 2)` holds of a list where every element is less than or equal to two. Recall that `z≤n` proves `zero ≤ n` for any `n`, and that if `m≤n` proves `m ≤ n` then `s≤s m≤n` proves `suc m ≤ suc n`, for any `m` and `n`:

```
_ : All (_≤ 2) [ 0 , 1 , 2 ]
_ = z≤n :: s≤s z≤n :: s≤s (s≤s z≤n) :: []
```

Here `_::_` and `[]` are the constructors of `All P` rather than of `List A`. The three items are proofs of `0 ≤ 2`, `1 ≤ 2`, and `2 ≤ 2`, respectively.

(One might wonder whether a pattern such as `[_,_,_]` can be used to construct values of type `All` as well as type `List`, since both use the same constructors. Indeed it can, so long as both types are in scope when the pattern is declared. That's not the case here, since `List` is defined before `[_,_,_]`, but `All` is defined later.)

Any

Predicate `Any P` holds if predicate `P` is satisfied by some element of a list:

```
data Any {A : Set} (P : A → Set) : List A → Set where
  here : ∀ {x : A} {xs : List A} → P x → Any P (x :: xs)
  there : ∀ {x : A} {xs : List A} → Any P xs → Any P (x :: xs)
```

The first constructor provides evidence that the head of the list satisfies `P`, while the second provides evidence that some element of the tail of the list satisfies `P`. For example, we can define list membership as follows:

```
infix 4 _∈_ _∉_

_∈_ : ∀ {A : Set} (x : A) (xs : List A) → Set
x ∈ xs = Any (x ≡_) xs

_∉_ : ∀ {A : Set} (x : A) (xs : List A) → Set
x ∉ xs = ¬ (x ∈ xs)
```

For example, zero is an element of the list `[ 0 , 1 , 0 , 2 ]`. Indeed, we can demonstrate this fact in two different ways, corresponding to the two different occurrences of zero in the list, as the first element and as the third element:

```
_ : 0 ∈ [ 0 , 1 , 0 , 2 ]
_ = here refl

_ : 0 ∈ [ 0 , 1 , 0 , 2 ]
_ = there (there (here refl))
```

Further, we can demonstrate that three is not in the list, because any possible proof that it is in the list leads to contradiction:

```
not-in : 3 ∉ [ 0 , 1 , 0 , 2 ]
not-in (here ())
not-in (there (here ()))
not-in (there (there (here ())))
not-in (there (there (there (here ()))))
not-in (there (there (there (there ()))))
```

The five occurrences of `()` attest to the fact that there is no possible evidence for `3 ≡ 0`, `3 ≡ 1`, `3 ≡ 0`, `3 ≡ 2`, and `3 ∈ []`, respectively.

All and append

A predicate holds for every element of one list appended to another if and only if it holds for every element of both lists:

```
All-++-⇔ : ∀ {A : Set} {P : A → Set} (xs ys : List A) →
  All P (xs ++ ys) ⇔ (All P xs × All P ys)
All-++-⇔ xs ys =
  record
    { to   = to xs ys
    ; from = from xs ys
    }
  where

  to : ∀ {A : Set} {P : A → Set} (xs ys : List A) →
    All P (xs ++ ys) → (All P xs × All P ys)
  to [] ys Pys = ⟨ [] , Pys ⟩
  to (x :: xs) ys (Px :: Pxs++ys) with to xs ys Pxs++ys
  ... | ⟨ Pxs , Pys ⟩ = ⟨ Px :: Pxs , Pys ⟩

  from : ∀ { A : Set} {P : A → Set} (xs ys : List A) →
    All P xs × All P ys → All P (xs ++ ys)
  from [] ys ⟨ [] , Pys ⟩ = Pys
  from (x :: xs) ys ⟨ Px :: Pxs , Pys ⟩ = Px :: from xs ys ⟨ Pxs , Pys ⟩
```

Exercise `Any-++-⇔` (recommended)

Prove a result similar to `All-++-⇔`, but with `Any` in place of `All`, and a suitable replacement for `_×_`. As a consequence, demonstrate an equivalence relating `_∈_` and `_++_`.

```
-- Your code goes here
```

Exercise `All-++-≃` (stretch)

Show that the equivalence `All-++-⇔` can be extended to an isomorphism.

```
-- Your code goes here
```

Exercise `¬Any⇔All¬` (recommended)

Show that `Any` and `All` satisfy a version of De Morgan's Law:

```
(¬_ ∘ Any P) xs ⇔ All (¬_ ∘ P) xs
```

(Can you see why it is important that here `_∘_` is generalised to arbitrary levels, as described in the section on universe polymorphism?)

Do we also have the following?

```
(¬_ ∘ All P) xs ⇔ Any (¬_ ∘ P) xs
```

If so, prove; if not, explain why.

```
-- Your code goes here
```

Exercise `¬Any≃All¬` (stretch)

Show that the equivalence `¬Any⇔All¬` can be extended to an isomorphism. You will need to use extensionality.

```
-- Your code goes here
```

Exercise `All-∀` (practice)

Show that `All P xs` is isomorphic to `∀ x → x ∈ xs → P x`.

```
-- You code goes here
```

Exercise `Any-∃` (practice)

Show that `Any P xs` is isomorphic to `∃[ x ] (x ∈ xs × P x)`.

```
-- You code goes here
```

Decidability of All

If we consider a predicate as a function that yields a boolean, it is easy to define an analogue of `All`, which returns true if a given predicate returns true for every element of a list:

```
all : ∀ {A : Set} → (A → Bool) → List A → Bool
all p = foldr _∧_ true ∘ map p
```

The function can be written in a particularly compact style by using the higher-order functions `map` and `foldr`.

As one would hope, if we replace booleans by decidables there is again an analogue of `All`. First, return to the notion of a predicate `P` as a function of type `A → Set`, taking a value `x` of type `A` into evidence `P x` that a property holds for `x`. Say that a predicate `P` is *decidable* if we have a function that for a given `x` can decide `P x`:

```
Decidable : ∀ {A : Set} → (A → Set) → Set
Decidable {A} P = ∀ (x : A) → Dec (P x)
```

Then if predicate `P` is decidable, it is also decidable whether every element of a list satisfies the predicate:

```
All? : ∀ {A : Set} {P : A → Set} → Decidable P → Decidable (All P)
All? P? []                                = yes []
All? P? (x :: xs) with P? x | All? P? xs
... | yes Px | yes Pxs                    = yes (Px :: Pxs)
... | no ¬Px | _                          = no λ{ (Px :: Pxs) → ¬Px Px }
... | _                     | no ¬Pxs     = no λ{ (Px :: Pxs) → ¬Pxs Pxs }
```

If the list is empty, then trivially `P` holds for every element of the list. Otherwise, the structure of the proof is similar to that showing that the conjunction of two decidable propositions is itself decidable, using `_::_` rather than `⟨_,_⟩` to combine the evidence for the head and tail of the list.

Exercise `Any?` (stretch)

Just as `All` has analogues `all` and `All?` which determine whether a predicate holds for every element of a list, so does `Any` have analogues `any` and `Any?` which determine whether a predicate holds for some element of a list. Give their definitions.

```
-- Your code goes here
```

Exercise `split` (stretch)

The relation `merge` holds when two lists merge to give a third list.

```
data merge {A : Set} : (xs ys zs : List A) → Set where

  [] :
      --------------
      merge [] [] []

  left-:: : ∀ {x xs ys zs}
    → merge xs ys zs
      --------------------------
    → merge (x :: xs) ys (x :: zs)

  right-:: : ∀ {y xs ys zs}
    → merge xs ys zs
      --------------------------
    → merge xs (y :: ys) (y :: zs)
```

For example,

```
_ : merge [ 1 , 4 ] [ 2 , 3 ] [ 1 , 2 , 3 , 4 ]
_ = left-:: (right-:: (right-:: (left-:: [])))
```

Given a decidable predicate and a list, we can split the list into two lists that merge to give the original list, where all elements of one list satisfy the predicate, and all elements of the other do not satisfy the predicate.

Define the following variant of the traditional `filter` function on lists, which given a decidable predicate and a list returns a list of elements that satisfy the predicate and a list of elements that don't, with their corresponding proofs.

```
split : ∀ {A : Set} {P : A → Set} (P? : Decidable P) (zs : List A)
  → ∃[ xs ] ∃[ ys ] ( merge xs ys zs × All P xs × All (¬_ ∘ P) ys )
```

```
-- Your code goes here
```

Standard Library

Definitions similar to those in this chapter can be found in the standard library:

```
import Data.List using (List; _++_; length; reverse; map; foldr; downFrom)
import Data.List.Relation.Unary.All using (All; []; _∷_)
import Data.List.Relation.Unary.Any using (Any; here; there)
import Data.List.Membership.Propositional using (_∈_)
import Data.List.Properties
  using (reverse-++-commute; map-compose; map-++-commute; foldr-++)
  renaming (mapIsFold to map-is-foldr)
import Algebra.Structures using (IsMonoid)
import Relation.Unary using (Decidable)
import Relation.Binary using (Decidable)
```

The standard library version of `IsMonoid` differs from the one given here, in that it is also parameterised on an equivalence relation.

Both `Relation.Unary` and `Relation.Binary` define a version of `Decidable`, one for unary relations (as used in this chapter where `P` ranges over unary predicates) and one for binary relations (as used earlier, where `_≤_` ranges over a binary relation).

Unicode

This chapter uses the following unicode:

```
∷  U+2237  PROPORTION  (\::)
⊗  U+2297  CIRCLED TIMES  (\otimes, \ox)
∈  U+2208  ELEMENT OF  (\in)
∉  U+2209  NOT AN ELEMENT OF  (\inn, \notin)
```

Part II

Part 2: Programming Language Foundations

Chapter 11

Lambda: Introduction to Lambda Calculus

```
module plfa.part2.Lambda where
```

The *lambda-calculus*, first published by the logician Alonzo Church in 1932, is a core calculus with only three syntactic constructs: variables, abstraction, and application. It captures the key concept of *functional abstraction*, which appears in pretty much every programming language, in the form of either functions, procedures, or methods. The *simply-typed lambda calculus* (or STLC) is a variant of the lambda calculus published by Church in 1940. It has the three constructs above for function types, plus whatever else is required for base types. Church had a minimal base type with no operations. We will instead echo Plotkin's *Programmable Computable Functions* (PCF), and add operations on natural numbers and recursive function definitions.

This chapter formalises the simply-typed lambda calculus, giving its syntax, small-step semantics, and typing rules. The next chapter Properties proves its main properties, including progress and preservation. Following chapters will look at a number of variants of lambda calculus.

Be aware that the approach we take here is *not* our recommended approach to formalisation. Using de Bruijn indices and intrinsically-typed terms, as we will do in Chapter DeBruijn, leads to a more compact formulation. Nonetheless, we begin with named variables and extrinsically-typed terms, partly because names are easier than indices to read, and partly because the development is more traditional.

The development in this chapter was inspired by the corresponding development in Chapter *Stlc* of *Software Foundations* (*Programming Language Foundations*). We differ by representing contexts explicitly (as lists pairing identifiers with types) rather than as partial maps (which take identifiers to types), which corresponds better to our subsequent development of DeBruijn notation. We also differ by taking natural numbers as the base type rather than booleans, allowing more sophisticated examples. In particular, we will be able to show (twice!) that two plus two is four.

Imports

```
open import Data.Bool using (T; not)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (List; _∷_; [])
open import Data.Nat using (ℕ; zero; suc)
open import Data.Product using (∃-syntax; _×_)
```

```
open import Data.String using (String; _≟_)
open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Relation.Nullary.Decidable using (⌊_⌋; False; toWitnessFalse)
open import Relation.Nullary.Negation using (¬?)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl)
```

Syntax of terms

Terms have seven constructs. Three are for the core lambda calculus:

- Variables `` ` x ``
- Abstractions `ƛ x ⇒ N`
- Applications `L · M`

Three are for the naturals:

- Zero `` `zero ``
- Successor `` `suc M ``
- Case `case L [zero⇒ M |suc x ⇒ N ]`

And one is for recursion:

- Fixpoint `μ x ⇒ M`

Abstraction is also called *lambda abstraction*, and is the construct from which the calculus takes its name.

With the exception of variables and fixpoints, each term form either constructs a value of a given type (abstractions yield functions, zero and successor yield natural numbers) or deconstructs it (applications use functions, case terms use naturals). We will see this again when we come to the rules for assigning types to terms, where constructors correspond to introduction rules and deconstructors to eliminators.

Here is the syntax of terms in Backus-Naur Form (BNF):

```
L, M, N  ::=
  ` x  |  ƛ x ⇒ N  |  L · M  |
  `zero  |  `suc M  |  case L [zero⇒ M |suc x ⇒ N ]  |
  μ x ⇒ M
```

And here it is formalised in Agda:

```
Id : Set
Id = String

infix 5  ƛ_⇒_
infix 5  μ_⇒_
infixl 7 _·_
infix 8  `suc_
infix 9  `_

data Term : Set where
```

```
`_                   : Id → Term
ƛ_⇒_                 : Id → Term → Term
_·_                  : Term → Term → Term
`zero                : Term
`suc_                : Term → Term
case_[zero⇒_|suc_⇒_] : Term → Term → Id → Term → Term
μ_⇒_                 : Id → Term → Term
```

We represent identifiers by strings. We choose precedence so that lambda abstraction and fixpoint bind least tightly, then application, then successor, and tightest of all is the constructor for variables. Case expressions are self-bracketing.

Example terms

Here are some example terms: the natural number two, a function that adds naturals, and a term that computes two plus two:

```
two : Term
two = `suc `suc `zero

plus : Term
plus = μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
        case ` "m"
          [zero⇒ ` "n"
          |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

The recursive definition of addition is similar to our original definition of `_+_` for naturals, as given in Chapter Naturals. Here variable "m" is bound twice, once in a lambda abstraction and once in the successor branch of the case; the first use of "m" refers to the former and the second to the latter. Any use of "m" in the successor branch must refer to the latter binding, and so we say that the latter binding *shadows* the former. Later we will confirm that two plus two is four, in other words that the term

```
plus · two · two
```

reduces to `` `suc `suc `suc `suc `zero ``.

As a second example, we use higher-order functions to represent natural numbers. In particular, the number *n* is represented by a function that accepts two arguments and applies the first *n* times to the second. This is called the *Church representation* of the naturals. Here are some example terms: the Church numeral two, a function that adds Church numerals, a function to compute successor, and a term that computes two plus two:

```
twoᶜ : Term
twoᶜ = ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")

plusᶜ : Term
plusᶜ = ƛ "m" ⇒ ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒
        ` "m" · ` "s" · (` "n" · ` "s" · ` "z")

sucᶜ : Term
sucᶜ = ƛ "n" ⇒ `suc (` "n")
```

The Church numeral for two takes two arguments `s` and `z` and applies `s` twice to `z`. Addition takes two numerals `m` and `n`, a function `s` and an argument `z`, and it uses `m` to apply `s` to the result of using `n` to apply `s` to `z`; hence `s` is applied `m` plus `n` times to `z`, yielding the Church numeral for the sum of `m` and `n`. For convenience, we define a function that computes

successor. To convert a Church numeral to the corresponding natural, we apply it to the `sucᶜ` function and the natural number zero. Again, later we will confirm that two plus two is four, in other words that the term

```
plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

reduces to `` `suc `suc `suc `suc `zero ``.

Exercise `mul` (recommended)

Write out the definition of a lambda term that multiplies two natural numbers. Your definition may use `plus` as defined earlier.

```
-- Your code goes here
```

Exercise `mulᶜ` (practice)

Write out the definition of a lambda term that multiplies two natural numbers represented as Church numerals. Your definition may use `plusᶜ` as defined earlier (or may not — there are nice definitions both ways).

```
-- Your code goes here
```

Exercise `primed` (stretch)

Some people find it annoying to write `` ` "x" `` instead of `x`. We can make examples with lambda terms slightly easier to write by adding the following definitions:

```
ƛ′_⇒_ : Term → Term → Term
ƛ′ (` x) ⇒ N = ƛ x ⇒ N
ƛ′ _ ⇒ _     = ⊥-elim impossible
  where postulate impossible : ⊥

case′_[zero⇒_|suc_⇒_] : Term → Term → Term → Term → Term
case′ L [zero⇒ M |suc (` x) ⇒ N ] = case L [zero⇒ M |suc x ⇒ N ]
case′ _ [zero⇒ _ |suc _ ⇒ _ ]     = ⊥-elim impossible
  where postulate impossible : ⊥

μ′_⇒_ : Term → Term → Term
μ′ (` x) ⇒ N = μ x ⇒ N
μ′ _ ⇒ _     = ⊥-elim impossible
  where postulate impossible : ⊥
```

We intend to apply the function only when the first term is a variable, which we indicate by postulating a term `impossible` of the empty type `⊥`. If we use C-c C-n to normalise the term

```
ƛ′ two ⇒ two
```

Agda will return an answer warning us that the impossible has occurred:

```
⊥-elim (plfa.part2.Lambda.impossible (`` `suc (`suc `zero)) (`suc (`suc `zero)) ``)
```

While postulating the impossible is a useful technique, it must be used with care, since such postulation could allow us to provide evidence of *any* proposition whatsoever, regardless of its truth.

The definition of `plus` can now be written as follows:

```
plus′ : Term
plus′ = μ′ + ⇒ ƛ′ m ⇒ ƛ′ n ⇒
         case′ m
           [zero⇒ n
           |suc m ⇒ `suc (+ · m · n) ]
  where
  + = ` "+"
  m = ` "m"
  n = ` "n"
```

Write out the definition of multiplication in the same style.

Formal vs informal

In informal presentation of formal semantics, one uses choice of variable name to disambiguate and writes `x` rather than `` ` x `` for a term that is a variable. Agda requires we distinguish.

Similarly, informal presentation often use the same notation for function types, lambda abstraction, and function application in both the *object language* (the language one is describing) and the *meta-language* (the language in which the description is written), trusting readers can use context to distinguish the two. Agda is not quite so forgiving, so here we use `ƛ x ⇒ N` and `L · M` for the object language, as compared to `λ x → N` and `L M` in our meta-language, Agda.

Bound and free variables

In an abstraction `ƛ x ⇒ N` we call `x` the *bound* variable and `N` the *body* of the abstraction. A central feature of lambda calculus is that consistent renaming of bound variables leaves the meaning of a term unchanged. Thus the five terms

- `ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")`
- `ƛ "f" ⇒ ƛ "x" ⇒ ` "f" · (` "f" · ` "x")`
- `ƛ "sam" ⇒ ƛ "zelda" ⇒ ` "sam" · (` "sam" · ` "zelda")`
- `ƛ "z" ⇒ ƛ "s" ⇒ ` "z" · (` "z" · ` "s")`
- `ƛ "😇" ⇒ ƛ "😈" ⇒ ` "😇" · (` "😇" · ` "😈")`

are all considered equivalent. Following the convention introduced by Haskell Curry, who used the Greek letter `α` (*alpha*) to label such rules, this equivalence relation is called *alpha renaming*.

As we descend from a term into its subterms, variables that are bound may become free. Consider the following terms:

- `ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")` has both `s` and `z` as bound variables.
- `ƛ "z" ⇒ ` "s" · (` "s" · ` "z")` has `z` bound and `s` free.

- `` ` "s" · (` "s" · ` "z") `` has both `s` and `z` as free variables.

We say that a term with no free variables is *closed*; otherwise it is *open*. Of the three terms above, the first is closed and the other two are open. We will focus on reduction of closed terms.

Different occurrences of a variable may be bound and free. In the term

```
(ƛ "x" ⇒ ` "x") · ` "x"
```

the inner occurrence of `x` is bound while the outer occurrence is free. By alpha renaming, the term above is equivalent to

```
(ƛ "y" ⇒ ` "y") · ` "x"
```

in which `y` is bound and `x` is free. A common convention, called the *Barendregt convention*, is to use alpha renaming to ensure that the bound variables in a term are distinct from the free variables, which can avoid confusions that may arise if bound and free variables have the same names.

Case and recursion also introduce bound variables, which are also subject to alpha renaming. In the term

```
μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
  case ` "m"
    [zero⇒ ` "n"
    |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

notice that there are two binding occurrences of `m`, one in the first line and one in the last line. It is equivalent to the following term,

```
μ "plus" ⇒ ƛ "x" ⇒ ƛ "y" ⇒
  case ` "x"
    [zero⇒ ` "y"
    |suc "x′" ⇒ `suc (` "plus" · ` "x′" · ` "y") ]
```

where the two binding occurrences corresponding to `m` now have distinct names, `x` and `x′`.

Values

A *value* is a term that corresponds to an answer. Thus, `` `suc `suc `suc `suc `zero `` is a value, while `plus · two · two` is not. Following convention, we treat all function abstractions as values; thus, `plus` by itself is considered a value.

The predicate `Value M` holds if term `M` is a value:

```
data Value : Term → Set where

  V-ƛ : ∀ {x N}
      ---------------
    → Value (ƛ x ⇒ N)

  V-zero :
      -----------
      Value `zero
```

```
V-suc : ∀ {V}
  → Value V
    --------------
  → Value (`suc V)
```

In what follows, we let `V` and `W` range over values.

Formal vs informal

In informal presentations of formal semantics, using `V` as the name of a metavariable is sufficient to indicate that it is a value. In Agda, we must explicitly invoke the `Value` predicate.

Other approaches

An alternative is not to focus on closed terms, to treat variables as values, and to treat `ƛ x ⇒ N` as a value only if `N` is a value. Indeed, this is how Agda normalises terms. We consider this approach in Chapter Untyped.

Substitution

The heart of lambda calculus is the operation of substituting one term for a variable in another term. Substitution plays a key role in defining the operational semantics of function application. For instance, we have

```
  (ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) · sucᶜ · `zero
⟶
  (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero
⟶
  sucᶜ · (sucᶜ · `zero)
```

where we substitute `sucᶜ` for `` ` "s" `` and `` `zero `` for `` ` "z" `` in the body of the function abstraction.

We write substitution as `N [ x := V ]`, meaning "substitute term `V` for free occurrences of variable `x` in term `N`", or, more compactly, "substitute `V` for `x` in `N`", or equivalently, "in `N` replace `x` by `V`". Substitution works if `V` is any closed term; it need not be a value, but we use `V` since in fact we usually substitute values.

Here are some examples:

- `(ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) ["s" := sucᶜ]` yields `ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")`.
- `(sucᶜ · (sucᶜ · ` "z")) ["z" := `zero ]` yields `sucᶜ · (sucᶜ · `zero)`.
- `(ƛ "x" ⇒ ` "y") ["y" := `zero ]` yields `ƛ "x" ⇒ `zero`.
- `(ƛ "x" ⇒ ` "x") ["x" := `zero ]` yields `ƛ "x" ⇒ ` "x"`.
- `(ƛ "y" ⇒ ` "y") ["x" := `zero ]` yields `ƛ "y" ⇒ ` "y"`.

In the last but one example, substituting `` `zero `` for `x` in `ƛ "x" ⇒ ` "x"` does *not* yield `ƛ "x" ⇒ `zero`, since `x` is bound in the lambda abstraction. The choice of bound names is irrelevant: both `ƛ "x" ⇒ ` "x"` and `ƛ "y" ⇒ ` "y"` stand for the identity function. One way to

think of this is that `x` within the body of the abstraction stands for a *different* variable than `x` outside the abstraction, they just happen to have the same name.

We will give a definition of substitution that is only valid when term substituted for the variable is closed. This is because substitution by terms that are *not* closed may require renaming of bound variables. For example:

- `(ƛ "x" ⇒ ` "x" · ` "y") [ "y" := ` "x" · `zero]` should not yield `(ƛ "x" ⇒ ` "x" · (` "x" · `zero))`.

Instead, we should rename the bound variable to avoid capture:

- `(ƛ "x" ⇒ ` "x" · ` "y") [ "y" := ` "x" · `zero ]` should yield `ƛ "x′" ⇒ ` "x′" · (` "x" · `zero)`.

Here `x′` is a fresh variable distinct from `x`. Formal definition of substitution with suitable renaming is considerably more complex, so we avoid it by restricting to substitution by closed terms, which will be adequate for our purposes.

Here is the formal definition of substitution by closed terms in Agda:

```
infix 9 _[_:=_]

_[_:=_] : Term → Id → Term → Term
(` x) [ y := V ] with x ≟ y
... | yes _       = V
... | no _        = ` x
(ƛ x ⇒ N) [ y := V ] with x ≟ y
... | yes _       = ƛ x ⇒ N
... | no _        = ƛ x ⇒ N [ y := V ]
(L · M) [ y := V ]  = L [ y := V ] · M [ y := V ]
(`zero) [ y := V ] = `zero
(`suc M) [ y := V ] = `suc M [ y := V ]
(case L [zero⇒ M |suc x ⇒ N ]) [ y := V ] with x ≟ y
... | yes _       = case L [ y := V ] [zero⇒ M [ y := V ] |suc x ⇒ N ]
... | no _        = case L [ y := V ] [zero⇒ M [ y := V ] |suc x ⇒ N [ y := V ] ]
(μ x ⇒ N) [ y := V ] with x ≟ y
... | yes _       = μ x ⇒ N
... | no _        = μ x ⇒ N [ y := V ]
```

Let's unpack the first three cases:

- For variables, we compare `y`, the substituted variable, with `x`, the variable in the term. If they are the same, we yield `V`, otherwise we yield `x` unchanged.
- For abstractions, we compare `y`, the substituted variable, with `x`, the variable bound in the abstraction. If they are the same, we yield the abstraction unchanged, otherwise we substitute inside the body.
- For application, we recursively substitute in the function and the argument.

Case expressions and recursion also have bound variables that are treated similarly to those in lambda abstractions. Otherwise we simply push substitution recursively into the subterms.

Examples

Here is confirmation that the examples above are correct:

```
_ : (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) [ "s" := sucᶜ ] ≡ ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")
_ = refl

_ : (sucᶜ · (sucᶜ · ` "z")) [ "z" := `zero ] ≡ sucᶜ · (sucᶜ · `zero)
_ = refl

_ : (ƛ "x" ⇒ ` "y") [ "y" := `zero ] ≡ ƛ "x" ⇒ `zero
_ = refl

_ : (ƛ "x" ⇒ ` "x") [ "x" := `zero ] ≡ ƛ "x" ⇒ ` "x"
_ = refl

_ : (ƛ "y" ⇒ ` "y") [ "x" := `zero ] ≡ ƛ "y" ⇒ ` "y"
_ = refl
```

Quiz

What is the result of the following substitution?

```
(ƛ "y" ⇒ ` "x" · (ƛ "x" ⇒ ` "x")) [ "x" := `zero ]
```

1. `(ƛ "y" ⇒ ` "x" · (ƛ "x" ⇒ ` "x"))`
2. `(ƛ "y" ⇒ ` "x" · (ƛ "x" ⇒ `zero))`
3. `(ƛ "y" ⇒ `zero · (ƛ "x" ⇒ ` "x"))`
4. `(ƛ "y" ⇒ `zero · (ƛ "x" ⇒ `zero))`

Exercise `_[_:=_]′` (stretch)

The definition of substitution above has three clauses (`ƛ`, `case`, and `μ`) that invoke a `with` clause to deal with bound variables. Rewrite the definition to factor the common part of these three clauses into a single function, defined by mutual recursion with substitution.

```
-- Your code goes here
```

Reduction

We give the reduction rules for call-by-value lambda calculus. To reduce an application, first we reduce the left-hand side until it becomes a value (which must be an abstraction); then we reduce the right-hand side until it becomes a value; and finally we substitute the argument for the variable in the abstraction.

In an informal presentation of the operational semantics, the rules for reduction of applications are written as follows:

```
L ⟶ L′
-------------- ξ-·₁
L · M ⟶ L′ · M

M ⟶ M′
-------------- ξ-·₂
V · M ⟶ V · M′

---------------------------- β-ƛ
(ƛ x ⇒ N) · V ⟶ N [ x := V ]
```

The Agda version of the rules below will be similar, except that universal quantifications are made explicit, and so are the predicates that indicate which terms are values.

The rules break into two sorts. Compatibility rules direct us to reduce some part of a term. We give them names starting with the Greek letter `ξ` (*xi*). Once a term is sufficiently reduced, it will consist of a constructor and a deconstructor, in our case `ƛ` and `·`, which reduces directly. We give them names starting with the Greek letter `β` (*beta*) and such rules are traditionally called *beta rules*.

A bit of terminology: A term that matches the left-hand side of a reduction rule is called a *redex*. In the redex `(ƛ x ⇒ N) · V`, we may refer to `x` as the *formal parameter* of the function, and `V` as the *actual parameter* of the function application. Beta reduction replaces the formal parameter by the actual parameter.

If a term is a value, then no reduction applies; conversely, if a reduction applies to a term then it is not a value. We will show in the next chapter that this exhausts the possibilities: every well-typed term either reduces or is a value.

For numbers, zero does not reduce and successor reduces the subterm. A case expression reduces its argument to a number, and then chooses the zero or successor branch as appropriate. A fixpoint replaces the bound variable by the entire fixpoint term; this is the one case where we substitute by a term that is not a value.

Here are the rules formalised in Agda:

```
infix 4 _⟶_

data _⟶_ : Term → Term → Set where

  ξ-·₁ : ∀ {L L′ M}
    → L ⟶ L′
      ------------------
    → L · M ⟶ L′ · M

  ξ-·₂ : ∀ {V M M′}
    → Value V
    → M ⟶ M′
      ------------------
    → V · M ⟶ V · M′

  β-ƛ : ∀ {x N V}
    → Value V
      ------------------------------
    → (ƛ x ⇒ N) · V ⟶ N [ x := V ]

  ξ-suc : ∀ {M M′}
    → M ⟶ M′
      ------------------
    → `suc M ⟶ `suc M′
```

```
ξ-case : ∀ {x L L′ M N}
  → L ⟶ L′
    ------------------------------------------------------------------
  → case L [zero⇒ M |suc x ⇒ N ] ⟶ case L′ [zero⇒ M |suc x ⇒ N ]

β-zero : ∀ {x M N}
    ---------------------------------------
  → case `zero [zero⇒ M |suc x ⇒ N ] ⟶ M

β-suc : ∀ {x V M N}
  → Value V
    ---------------------------------------------------
  → case `suc V [zero⇒ M |suc x ⇒ N ] ⟶ N [ x := V ]

β-μ : ∀ {x M}
    ------------------------------
  → μ x ⇒ M ⟶ M [ x := μ x ⇒ M ]
```

The reduction rules are carefully designed to ensure that subterms of a term are reduced to values before the whole term is reduced. This is referred to as *call-by-value* reduction.

Further, we have arranged that subterms are reduced in a left-to-right order. This means that reduction is *deterministic*: for any term, there is at most one other term to which it reduces. Put another way, our reduction relation `⟶` is in fact a function.

This style of explaining the meaning of terms is called a *small-step operational semantics*. If `M ⟶ N`, we say that term `M` *reduces* to term `N`, or equivalently, term `M` *steps* to term `N`. Each compatibility rule has another reduction rule in its premise; so a step always consists of a beta rule, possibly adjusted by zero or more compatibility rules.

Quiz

What does the following term step to?

```
(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")  ⟶  ???
```

1. `(ƛ "x" ⇒ ` "x")`
2. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`
3. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`

What does the following term step to?

```
(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")  ⟶  ???
```

1. `(ƛ "x" ⇒ ` "x")`
2. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`
3. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`

What does the following term step to? (Where `twoᶜ` and `sucᶜ` are as defined above.)

```
twoᶜ · sucᶜ · `zero  ⟶  ???
```

1. `sucᶜ · (sucᶜ · `zero)`
2. `(ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero`
3. `` `zero ``

Reflexive and transitive closure

A single step is only part of the story. In general, we wish to repeatedly step a closed term until it reduces to a value. We do this by defining the reflexive and transitive closure `—↠` of the step relation `—→`.

We define reflexive and transitive closure as a sequence of zero or more steps of the underlying relation, along lines similar to that for reasoning about chains of equalities in Chapter Equality:

```
infix 2 _—↠_
infix 1 begin_
infixr 2 _—→⟨_⟩_
infix 3 _∎

data _—↠_ : Term → Term → Set where
  _∎ : ∀ M
     ---------
    → M —↠ M

  _—→⟨_⟩_ : ∀ L {M N}
    → L —→ M
    → M —↠ N
      ---------
    → L —↠ N

begin_ : ∀ {M N}
  → M —↠ N
    ------
  → M —↠ N
begin M—↠N = M—↠N
```

We can read this as follows:

- From term `M`, we can take no steps, giving a step of type `M —↠ M`. It is written `M ∎`.
- From term `L` we can take a single step of type `L —→ M` followed by zero or more steps of type `M —↠ N`, giving a step of type `L —↠ N`. It is written `L —→⟨ L—→M ⟩ M—↠N`, where `L—→M` and `M—↠N` are steps of the appropriate type.

The notation is chosen to allow us to lay out example reductions in an appealing way, as we will see in the next section.

An alternative is to define reflexive and transitive closure directly, as the smallest relation that includes `—→` and is also reflexive and transitive. We could do so as follows:

```
data _—↠′_ : Term → Term → Set where
  step′ : ∀ {M N}
    → M —→ N
      -------
    → M —↠′ N
  refl′ : ∀ {M}
```

```
    -------
  → M —↠′ M

trans′ : ∀ {L M N}
  → L —↠′ M
  → M —↠′ N
    -------
  → L —↠′ N
```

The three constructors specify, respectively, that `—↠′` includes `⟶` and is reflexive and transitive. A good exercise is to show that the two definitions are equivalent (indeed, one embeds in the other).

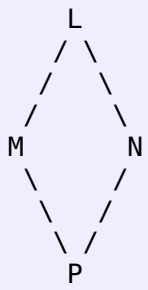
Exercise `—↠≲—↠′` (practice)

Show that the first notion of reflexive and transitive closure above embeds into the second. Why are they not isomorphic?

```
-- Your code goes here
```

Confluence

One important property a reduction relation might satisfy is to be *confluent*. If term `L` reduces to two other terms, `M` and `N`, then both of these reduce to a common term `P`. It can be illustrated as follows:



Here `L`, `M`, `N` are universally quantified while `P` is existentially quantified. If each line stands for zero or more reduction steps, this is called confluence, while if the top two lines stand for a single reduction step and the bottom two stand for zero or more reduction steps it is called the diamond property. In symbols:

```
postulate
  confluence : ∀ {L M N}
    → ((L —↠ M) × (L —↠ N))
      ----------------------
    → ∃[ P ] ((M —↠ P) × (N —↠ P))

  diamond : ∀ {L M N}
    → ((L ⟶ M) × (L ⟶ N))
      --------------------
    → ∃[ P ] ((M —↠ P) × (N —↠ P))
```

The reduction system studied in this chapter is deterministic. In symbols:

```
postulate
  deterministic : ∀ {L M N}
    → L —→ M
    → L —→ N
      ------
    → M ≡ N
```

It is easy to show that every deterministic relation satisfies the diamond and confluence properties. Hence, all the reduction systems studied in this text are trivially confluent.

Examples

We start with a simple example. The Church numeral two applied to the successor function and zero yields the natural number two:

```
_ : twoᶜ · sucᶜ · `zero —↠ `suc `suc `zero
_ =
  begin
    twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero
  —→⟨ β-ƛ V-zero ⟩
    sucᶜ · (sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    sucᶜ · `suc `zero
  —→⟨ β-ƛ (V-suc V-zero) ⟩
    `suc (`suc `zero)
  ∎
```

Here is a sample reduction demonstrating that two plus two is four:

```
_ : plus · two · two —↠ `suc `suc `suc `suc `zero
_ =
  begin
    plus · two · two
  —→⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
    (ƛ "m" ⇒ ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two · two
  —→⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ "n" ⇒
      case two [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    case two [zero⇒ two |suc "m" ⇒ `suc (plus · ` "m" · two) ]
  —→⟨ β-suc (V-suc V-zero) ⟩
    `suc (plus · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
    `suc ((ƛ "m" ⇒ ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
    `suc ((ƛ "n" ⇒
      case `suc `zero [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two)
```

```
  —→⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
    `suc (case `suc `zero [zero⇒ two |suc "m" ⇒ `suc (plus · ` "m" · two) ])
  —→⟨ ξ-suc (β-suc V-zero) ⟩
    `suc `suc (plus · `zero · two)
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
    `suc `suc ((ƛ "m" ⇒ ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · `zero · two)
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
    `suc `suc ((ƛ "n" ⇒
      case `zero [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two)
  —→⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
    `suc `suc (case `zero [zero⇒ two |suc "m" ⇒ `suc (plus · ` "m" · two) ])
  —→⟨ ξ-suc (ξ-suc β-zero) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

And here is a similar sample reduction for Church numerals:

```
_ : plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero —↠ `suc `suc `suc `suc `zero
_ =
  begin
    (ƛ "m" ⇒ ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒ ` "m" · ` "s" · (` "n" · ` "s" · ` "z"))
      · twoᶜ · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
    (ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒ twoᶜ · ` "s" · (` "n" · ` "s" · ` "z"))
      · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ "s" ⇒ ƛ "z" ⇒ twoᶜ · ` "s" · (twoᶜ · ` "s" · ` "z")) · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ twoᶜ · sucᶜ · (twoᶜ · sucᶜ · ` "z")) · `zero
  —→⟨ β-ƛ V-zero ⟩
    twoᶜ · sucᶜ · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · ((ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (sucᶜ · (sucᶜ · `zero))
  —→⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (sucᶜ · (`suc `zero))
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (`suc `suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    sucᶜ · (sucᶜ · `suc `suc `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    sucᶜ · (`suc `suc `suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

In the next chapter, we will see how to compute such reduction sequences.

Exercise `plus-example` **(practice)**

Write out the reduction sequence demonstrating that one plus one is two.

```
-- Your code goes here
```

Syntax of types

We have just two types:

- Functions, `A ⇒ B`
- Naturals, `` `ℕ ``

As before, to avoid overlap we use variants of the names used by Agda.

Here is the syntax of types in BNF:

```
A, B, C  ::=  A ⇒ B | `ℕ
```

And here it is formalised in Agda:

```
infixr 7 _⇒_

data Type : Set where
  _⇒_ : Type → Type → Type
  `ℕ : Type
```

Precedence

As in Agda, functions of two or more arguments are represented via currying. This is made more convenient by declaring `_⇒_` to associate to the right and `_·_` to associate to the left. Thus:

- `(`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ` stands for `((`ℕ ⇒ `ℕ) ⇒ (`ℕ ⇒ `ℕ))`.
- `plus · two · two` stands for `(plus · two) · two`.

Quiz

- What is the type of the following term?

  `ƛ "s" ⇒ ` "s" · (` "s"  · `zero)`

  1. `(`ℕ ⇒ `ℕ) ⇒ (`ℕ ⇒ `ℕ)`
  2. `(`ℕ ⇒ `ℕ) ⇒ `ℕ`
  3. `` `ℕ ⇒ (`ℕ ⇒ `ℕ) ``
  4. `` `ℕ ⇒ `ℕ ⇒ `ℕ ``
  5. `` `ℕ ⇒ `ℕ ``
  6. `` `ℕ ``

  Give more than one answer if appropriate.

- What is the type of the following term?

  `(ƛ "s" ⇒ ` "s" · (` "s"  · `zero)) · sucᶜ`

1. `(`ℕ ⇒ `ℕ) ⇒ (`ℕ ⇒ `ℕ)`
2. `(`ℕ ⇒ `ℕ) ⇒ `ℕ`
3. ``ℕ ⇒ (`ℕ ⇒ `ℕ)`
4. `` `ℕ ⇒ `ℕ ⇒ `ℕ ``
5. `` `ℕ ⇒ `ℕ ``
6. `` `ℕ ``

Give more than one answer if appropriate.

Typing

Contexts

While reduction considers only closed terms, typing must consider terms with free variables. To type a term, we must first type its subterms, and in particular in the body of an abstraction its bound variable may appear free.

A *context* associates variables with types. We let `Γ` and `Δ` range over contexts. We write `∅` for the empty context, and `Γ , x ⦂ A` for the context that extends `Γ` by mapping variable `x` to type `A`. For example,

- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ`

is the context that associates variable `"s"` with type `` `ℕ ⇒ `ℕ ``, and variable `"z"` with type `` `ℕ ``.

Contexts are formalised as follows:

```
infixl 5 _,_⦂_

data Context : Set where
  ∅ : Context
  _,_⦂_ : Context → Id → Type → Context
```

Exercise `Context-≃` (practice)

Show that `Context` is isomorphic to `List (Id × Type)`. For instance, the isomorphism relates the context

```
∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ
```

to the list

```
[ ⟨ "z" , `ℕ ⟩ , ⟨ "s" , `ℕ ⇒ `ℕ ⟩ ]
```

```
-- Your code goes here
```

Lookup judgment

We have two forms of *judgment*. The first is written

```
Γ ∋ x ⦂ A
```

and indicates in context `Γ` that variable `x` has type `A`. It is called *lookup*. For example,

- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "s" ⦂ `ℕ ⇒ `ℕ`

give us the types associated with variables `"z"` and `"s"`, respectively. The symbol `∋` (pronounced "ni", for "in" backwards) is chosen because checking that `Γ ∋ x ⦂ A` is analogous to checking whether `x ⦂ A` appears in a list corresponding to `Γ`.

If two variables in a context have the same name, then lookup should return the most recently bound variable, which *shadows* the other variables. For example,

- `∅ , "x" ⦂ `ℕ ⇒ `ℕ , "x" ⦂ `ℕ ∋ "x" ⦂ `ℕ`.

Here `"x" ⦂ `ℕ ⇒ `ℕ` is shadowed by `"x" ⦂ `ℕ`.

Lookup is formalised as follows:

```
infix 4 _∋_⦂_

data _∋_⦂_ : Context → Id → Type → Set where

  Z : ∀ {Γ x A}
    ------------------
    → Γ , x ⦂ A ∋ x ⦂ A

  S : ∀ {Γ x y A B}
    → x ≢ y
    → Γ ∋ x ⦂ A
      ------------------
    → Γ , y ⦂ B ∋ x ⦂ A
```

The constructors `Z` and `S` correspond roughly to the constructors `here` and `there` for the element-of relation `_∈_` on lists. Constructor `S` takes an additional parameter, which ensures that when we look up a variable that it is not *shadowed* by another variable with the same name to its left in the list.

It can be rather tedious to use the `S` constructor, as you have to provide proofs that `x ≢ y` each time. For example:

```
_ : ∅ , "x" ⦂ `ℕ ⇒ `ℕ , "y" ⦂ `ℕ , "z" ⦂ `ℕ ∋ "x" ⦂ `ℕ ⇒ `ℕ
_ = S (λ()) (S (λ()) Z)
```

Instead, we'll use a "smart constructor", which uses proof by reflection to check the inequality while type checking:

```
S′ : ∀ {Γ x y A B}
   → {x≢y : False (x ≟ y)}
```

```
  → Γ ∋ x ⦂ A
    ------------------
  → Γ , y ⦂ B ∋ x ⦂ A

S′ {x≠y = x≠y} x = S (toWitnessFalse x≠y) x
```

Typing judgment

The second judgment is written

```
Γ ⊢ M ⦂ A
```

and indicates in context `Γ` that term `M` has type `A`. Context `Γ` provides types for all the free variables in `M`. For example:

- `ø , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "z" ⦂ `ℕ`
- `ø , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" ⦂ `ℕ ⇒ `ℕ`
- `ø , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" · ` "z" ⦂ `ℕ`
- `ø , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" · (` "s" · ` "z") ⦂ `ℕ`
- `ø , "s" ⦂ `ℕ ⇒ `ℕ ⊢ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") ⦂ `ℕ ⇒ `ℕ`
- `ø ⊢ ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") ⦂ (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ`

Typing is formalised as follows:

```
infix 4 _⊢_⦂_

data _⊢_⦂_ : Context → Term → Type → Set where

  -- Axiom
  ⊢` : ∀ {Γ x A}
    → Γ ∋ x ⦂ A
      -----------
    → Γ ⊢ ` x ⦂ A

  -- ⇒-I
  ⊢ƛ : ∀ {Γ x N A B}
    → Γ , x ⦂ A ⊢ N ⦂ B
      -------------------
    → Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B

  -- ⇒-E
  _·_ : ∀ {Γ L M A B}
    → Γ ⊢ L ⦂ A ⇒ B
    → Γ ⊢ M ⦂ A
      -------------
    → Γ ⊢ L · M ⦂ B

  -- ℕ-I₁
  ⊢zero : ∀ {Γ}
      --------------
    → Γ ⊢ `zero ⦂ `ℕ

  -- ℕ-I₂
  ⊢suc : ∀ {Γ M}
    → Γ ⊢ M ⦂ `ℕ
```

```
    ---------------
  → Γ ⊢ `suc M ⦂ `ℕ

-- ℕ-E
⊢case : ∀ {Γ L M x N A}
  → Γ ⊢ L ⦂ `ℕ
  → Γ ⊢ M ⦂ A
  → Γ , x ⦂ `ℕ ⊢ N ⦂ A
    ------------------------------------
  → Γ ⊢ case L [zero⇒ M |suc x ⇒ N ] ⦂ A

⊢μ : ∀ {Γ x M A}
  → Γ , x ⦂ A ⊢ M ⦂ A
    -----------------
  → Γ ⊢ μ x ⇒ M ⦂ A
```

Each type rule is named after the constructor for the corresponding term.

Most of the rules have a second name, derived from a convention in logic, whereby the rule is named after the type connective that it concerns; rules to introduce and to eliminate each connective are labeled `-I` and `-E`, respectively. As we read the rules from top to bottom, introduction and elimination rules do what they say on the tin: the first *introduces* a formula for the connective, which appears in the conclusion but not in the premises; while the second *eliminates* a formula for the connective, which appears in a premise but not in the conclusion. An introduction rule describes how to construct a value of the type (abstractions yield functions, successor and zero yield naturals), while an elimination rule describes how to deconstruct a value of the given type (applications use functions, case expressions use naturals).

Note also the three places (in `⊢ƛ`, `⊢case`, and `⊢μ`) where the context is extended with `x` and an appropriate type, corresponding to the three places where a bound variable is introduced.

The rules are deterministic, in that at most one rule applies to every term.

Example type derivations

Type derivations correspond to trees. In informal notation, here is a type derivation for the Church numeral two,

```
                        ∋s                    ∋z
                        ----------------- ⊢`  -------------- ⊢`
∋s                      Γ₂ ⊢ ` "s" ⦂ A ⇒ A    Γ₂ ⊢ ` "z" ⦂ A
----------------- ⊢`    ------------------------------------ _·_
Γ₂ ⊢ ` "s" ⦂ A ⇒ A      Γ₂ ⊢ ` "s" · ` "z" ⦂ A
------------------------------------------- _·_
Γ₂ ⊢ ` "s" · (` "s" · ` "z") ⦂ A
---------------------------------------- ⊢ƛ
Γ₁ ⊢ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") ⦂ A ⇒ A
------------------------------------------------------------ ⊢ƛ
Γ ⊢ ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") ⦂ (A ⇒ A) ⇒ A ⇒ A
```

where `∋s` and `∋z` abbreviate the two derivations,

```
             ---------------- Z
"s" ≠ "z"    Γ₁ ∋ "s" ⦂ A ⇒ A
---------------------------- S       ------------- Z
Γ₂ ∋ "s" ⦂ A ⇒ A                      Γ₂ ∋ "z" ⦂ A
```

and where `Γ₁ = Γ , "s" ⦂ A ⇒ A` and `Γ₂ = Γ , "s" ⦂ A ⇒ A , "z" ⦂ A`. The typing derivation is valid for any `Γ` and `A`, for instance, we might take `Γ` to be `∅` and `A` to be `` `ℕ ``.

Here is the above typing derivation formalised in Agda:

```
Ch : Type → Type
Ch A = (A ⇒ A) ⇒ A ⇒ A

⊢twoᶜ : ∀ {Γ A} → Γ ⊢ twoᶜ ⦂ Ch A
⊢twoᶜ = ⊢ƛ (⊢ƛ (⊢` ∋s · (⊢` ∋s · ⊢` ∋z)))
  where
  ∋s = S′ Z
  ∋z = Z
```

Here are the typings corresponding to computing two plus two:

```
⊢two : ∀ {Γ} → Γ ⊢ two ⦂ `ℕ
⊢two = ⊢suc (⊢suc ⊢zero)

⊢plus : ∀ {Γ} → Γ ⊢ plus ⦂ `ℕ ⇒ `ℕ ⇒ `ℕ
⊢plus = ⊢μ (⊢ƛ (⊢ƛ (⊢case (⊢` ∋m) (⊢` ∋n)
          (⊢suc (⊢` ∋+ · ⊢` ∋m′ · ⊢` ∋n′))))))
  where
  ∋+ = S′ (S′ (S′ Z))
  ∋m = S′ Z
  ∋n = Z
  ∋m′ = Z
  ∋n′ = S′ Z

⊢2+2 : ∅ ⊢ plus · two · two ⦂ `ℕ
⊢2+2 = ⊢plus · ⊢two · ⊢two
```

In contrast to our earlier examples, here we have typed `two` and `plus` in an arbitrary context rather than the empty context; this makes it easy to use them inside other binding contexts as well as at the top level. Here the two lookup judgments `∋m` and `∋m′` refer to two different bindings of variables named `"m"`. In contrast, the two judgments `∋n` and `∋n′` both refer to the same binding of `"n"` but accessed in different contexts, the first where `"n"` is the last binding in the context, and the second after `"m"` is bound in the successor branch of the case.

And here are typings for the remainder of the Church example:

```
⊢plusᶜ : ∀ {Γ A} → Γ ⊢ plusᶜ ⦂ Ch A ⇒ Ch A ⇒ Ch A
⊢plusᶜ = ⊢ƛ (⊢ƛ (⊢ƛ (⊢ƛ (⊢` ∋m · ⊢` ∋s · (⊢` ∋n · ⊢` ∋s · ⊢` ∋z)))))
  where
  ∋m = S′ (S′ (S′ Z))
  ∋n = S′ (S′ Z)
  ∋s = S′ Z
  ∋z = Z

⊢sucᶜ : ∀ {Γ} → Γ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` ∋n))
  where
  ∋n = Z

⊢2+2ᶜ : ∅ ⊢ plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero ⦂ `ℕ
⊢2+2ᶜ = ⊢plusᶜ · ⊢twoᶜ · ⊢twoᶜ · ⊢sucᶜ · ⊢zero
```

Interaction with Agda

Construction of a type derivation may be done interactively. Start with the declaration:

```
⊢sucᶜ : ∅ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
⊢sucᶜ = ?
```

Typing C-c C-l causes Agda to create a hole and tell us its expected type:

```
⊢sucᶜ = { }0
?0 : ∅ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
```

Now we fill in the hole by typing C-c C-r. Agda observes that the outermost term in `sucᶜ` is `ƛ`, which is typed using `⊢ƛ`. The `⊢ƛ` rule in turn takes one argument, which Agda leaves as a hole:

```
⊢sucᶜ = ⊢ƛ { }1
?1 : ∅ , "n" ⦂ `ℕ ⊢ `suc ` "n" ⦂ `ℕ
```

We can fill in the hole by typing C-c C-r again:

```
⊢sucᶜ = ⊢ƛ (⊢suc { }2)
?2 : ∅ , "n" ⦂ `ℕ ⊢ ` "n" ⦂ `ℕ
```

And again:

```
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` { }3))
?3 : ∅ , "n" ⦂ `ℕ ∋ "n" ⦂ `ℕ
```

A further attempt with C-c C-r yields the message:

```
Don't know which constructor to introduce of Z or S
```

We can fill in `Z` by hand. If we type C-c C-space, Agda will confirm we are done:

```
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` Z))
```

The entire process can be automated using Agsy, invoked with C-c C-a.

Chapter Inference will show how to use Agda to compute type derivations directly.

Lookup is injective

The lookup relation `Γ ∋ x ⦂ A` is injective, in that for each `Γ` and `x` there is at most one `A` such that the judgment holds:

```
∋-injective : ∀ {Γ x A B} → Γ ∋ x ⦂ A → Γ ∋ x ⦂ B → A ≡ B
∋-injective Z Z                 = refl
∋-injective Z (S x≠ _)          = ⊥-elim (x≠ refl)
∋-injective (S x≠ _) Z          = ⊥-elim (x≠ refl)
∋-injective (S _ ∋x) (S _ ∋x′) = ∋-injective ∋x ∋x′
```

The typing relation `Γ ⊢ M ⦂ A` is not injective. For example, in any `Γ` the term `ƛ "x" ⇒ ` "x"` has type `A ⇒ A` for any type `A`.

Non-examples

We can also show that terms are *not* typeable. For example, here is a formal proof that it is not possible to type the term `` `zero · `suc `zero `` . It cannot be typed, because doing so requires that the first term in the application is both a natural and a function:

```
nope₁ : ∀ {A} → ¬ (∅ ⊢ `zero · `suc `zero ⦂ A)
nope₁ (() · _)
```

As a second example, here is a formal proof that it is not possible to type `ƛ "x" ⇒ ` "x" · ` "x"` . It cannot be typed, because doing so requires types `A` and `B` such that `A ⇒ B ≡ A` :

```
nope₂ : ∀ {A} → ¬ (∅ ⊢ ƛ "x" ⇒ ` "x" · ` "x" ⦂ A)
nope₂ (⊢ƛ (⊢` ∋x · ⊢` ∋x′)) = contradiction (∋-injective ∋x ∋x′)
  where
  contradiction : ∀ {A B} → ¬ (A ⇒ B ≡ A)
  contradiction ()
```

Quiz

For each of the following, give a type `A` for which it is derivable, or explain why there is no such `A` .

1. `∅ , "y" ⦂ `ℕ ⇒ `ℕ , "x" ⦂ `ℕ ⊢ ` "y" · ` "x" ⦂ A`
2. `∅ , "y" ⦂ `ℕ ⇒ `ℕ , "x" ⦂ `ℕ ⊢ ` "x" · ` "y" ⦂ A`
3. `∅ , "y" ⦂ `ℕ ⇒ `ℕ ⊢ ƛ "x" ⇒ ` "y" · ` "x" ⦂ A`

For each of the following, give types `A` , `B` , and `C` for which it is derivable, or explain why there are no such types.

1. `∅ , "x" ⦂ A ⊢ ` "x" · ` "x" ⦂ B`
2. `∅ , "x" ⦂ A , "y" ⦂ B ⊢ ƛ "z" ⇒ ` "x" · (` "y" · ` "z") ⦂ C`

Exercise `⊢mul` (recommended)

Using the term `mul` you defined earlier, write out the derivation showing that it is well typed.

```
-- Your code goes here
```

Exercise `⊢mulᶜ` (practice)

Using the term `mulᶜ` you defined earlier, write out the derivation showing that it is well typed.

```
-- Your code goes here
```

Unicode

This chapter uses the following unicode:

```
⇒  U+21D2  RIGHTWARDS DOUBLE ARROW (\=>)
ƛ  U+019B  LATIN SMALL LETTER LAMBDA WITH STROKE (\Gl-)
·  U+00B7  MIDDLE DOT (\cdot)
≟  U+225F  QUESTIONED EQUAL TO (\?=)
—  U+2014  EM DASH (\em)
↠  U+21A0  RIGHTWARDS TWO HEADED ARROW (\rr-)
ξ  U+03BE  GREEK SMALL LETTER XI (\Gx or \xi)
β  U+03B2  GREEK SMALL LETTER BETA (\Gb or \beta)
Γ  U+0393  GREEK CAPITAL LETTER GAMMA (\GG or \Gamma)
≠  U+2260  NOT EQUAL TO (\=n or \ne)
∋  U+220B  CONTAINS AS MEMBER (\ni)
∅  U+2205  EMPTY SET (\0)
⊢  U+22A2  RIGHT TACK (\vdash or \|-)
⦂  U+2982  Z NOTATION TYPE COLON (\:)
😇  U+1F607  SMILING FACE WITH HALO
😈  U+1F608  SMILING FACE WITH HORNS
```

We compose reduction `⟶` from an em dash `—` and an arrow `→`. Similarly for reflexive and transitive closure `⟶↠`.

Chapter 12

Properties: Progress and Preservation

```
module plfa.part2.Properties where
```

This chapter covers properties of the simply-typed lambda calculus, as introduced in the previous chapter. The most important of these properties are progress and preservation. We introduce these below, and show how to combine them to get Agda to compute reduction sequences for us.

Imports

```
open import Relation.Binary.PropositionalEquality
  using (_≡_; _≢_; refl; sym; cong; cong₂)
open import Data.String using (String; _≟_)
open import Data.Nat using (ℕ; zero; suc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product
  using (_×_; proj₁; proj₂; ∃; ∃-syntax)
  renaming (_,_ to ⟨_,_⟩)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Relation.Nullary using (¬_; Dec; yes; no)
open import Function using (_∘_)
open import plfa.part1.Isomorphism
open import plfa.part2.Lambda
```

Introduction

The last chapter introduced simply-typed lambda calculus, including the notions of closed terms, terms that are values, reducing one term to another, and well-typed terms.

Ultimately, we would like to show that we can keep reducing a term until we reach a value. For instance, in the last chapter we showed that two plus two is four,

```
plus · two · two  —↠  `suc `suc `suc `suc `zero
```

which was proved by a long chain of reductions, ending in the value on the right. Every term in the chain had the same type, `` `ℕ ``. We also saw a second, similar example involving Church numerals.

What we might expect is that every term is either a value or can take a reduction step. As we will see, this property does *not* hold for every term, but it does hold for every closed, well-typed term.

Progress: If `∅ ⊢ M ⦂ A` then either `M` is a value or there is an `N` such that `M ⟶ N`.

So, either we have a value, and we are done, or we can take a reduction step. In the latter case, we would like to apply progress again. But to do so we need to know that the term yielded by the reduction is itself closed and well typed. It turns out that this property holds whenever we start with a closed, well-typed term.

Preservation: If `∅ ⊢ M ⦂ A` and `M ⟶ N` then `∅ ⊢ N ⦂ A`.

This gives us a recipe for automating evaluation. Start with a closed and well-typed term. By progress, it is either a value, in which case we are done, or it reduces to some other term. By preservation, that other term will itself be closed and well typed. Repeat. We will either loop forever, in which case evaluation does not terminate, or we will eventually reach a value, which is guaranteed to be closed and of the same type as the original term. We will turn this recipe into Agda code that can compute for us the reduction sequence of `plus · two · two`, and its Church numeral variant.

(The development in this chapter was inspired by the corresponding development in *Software Foundations*, Volume *Programming Language Foundations*, Chapter *StlcProp*. It will turn out that one of our technical choices — to introduce an explicit judgment `Γ ∋ x ⦂ A` in place of treating a context as a function from identifiers to types — permits a simpler development. In particular, we can prove substitution preserves types without needing to develop a separate inductive definition of the `appears_free_in` relation.)

Values do not reduce

We start with an easy observation. Values do not reduce:

```
V¬⟶ : ∀ {M N}
  → Value M
    ----------
  → ¬ (M ⟶ N)
V¬⟶ V-ƛ    ()
V¬⟶ V-zero ()
V¬⟶ (V-suc VM) (ξ-suc M⟶N) = V¬⟶ VM M⟶N
```

We consider the three possibilities for values:

- If it is an abstraction then no reduction applies
- If it is zero then no reduction applies
- If it is a successor then rule `ξ-suc` may apply, but in that case the successor is itself of a value that reduces, which by induction cannot occur.

As a corollary, terms that reduce are not values:

```
—→¬V : ∀ {M N}
  → M —→ N
    ---------
  → ¬ Value M
—→¬V M—→N VM = V¬—→ VM M—→N
```

If we expand out the negations, we have

```
V¬—→ : ∀ {M N} → Value M → M —→ N → ⊥
—→¬V : ∀ {M N} → M —→ N → Value M → ⊥
```

which are the same function with the arguments swapped.

Canonical Forms

Well-typed values must take one of a small number of *canonical forms*, which provide an analogue of the `Value` relation that relates values to their types. A lambda expression must have a function type, and a zero or successor expression must be a natural. Further, the body of a function must be well typed in a context containing only its bound variable, and the argument of successor must itself be canonical:

```
infix 4 Canonical_⦂_

data Canonical_⦂_ : Term → Type → Set where

  C-ƛ : ∀ {x A N B}
    → ∅ , x ⦂ A ⊢ N ⦂ B
      ------------------------------
    → Canonical (ƛ x ⇒ N) ⦂ (A ⇒ B)

  C-zero :
      --------------------
      Canonical `zero ⦂ `ℕ

  C-suc : ∀ {V}
    → Canonical V ⦂ `ℕ
      ---------------------
    → Canonical `suc V ⦂ `ℕ
```

Every closed, well-typed value is canonical:

```
canonical : ∀ {V A}
  → ∅ ⊢ V ⦂ A
  → Value V
    -----------
  → Canonical V ⦂ A
canonical (⊢` ())   ()
canonical (⊢ƛ ⊢N)   V-ƛ        = C-ƛ ⊢N
canonical (⊢L · ⊢M) ()
canonical ⊢zero     V-zero     = C-zero
canonical (⊢suc ⊢V) (V-suc VV) = C-suc (canonical ⊢V VV)
canonical (⊢case ⊢L ⊢M ⊢N) ()
canonical (⊢μ ⊢M)   ()
```

There are only three interesting cases to consider:

- If the term is a lambda abstraction, then well-typing of the term guarantees well-typing of the body.
- If the term is zero then it is canonical trivially.
- If the term is a successor then since it is well typed its argument is well typed, and since it is a value its argument is a value. Hence, by induction its argument is also canonical.

The variable case is thrown out because a closed term has no free variables and because a variable is not a value. The cases for application, case expression, and fixpoint are thrown out because they are not values.

Conversely, if a term is canonical then it is a value and it is well typed in the empty context:

```
value : ∀ {M A}
  → Canonical M ⦂ A
    ----------------
  → Value M
value (C-ƛ ⊢N)   = V-ƛ
value C-zero     = V-zero
value (C-suc CM) = V-suc (value CM)

typed : ∀ {M A}
  → Canonical M ⦂ A
    ---------------
  → ∅ ⊢ M ⦂ A
typed (C-ƛ ⊢N)   = ⊢ƛ ⊢N
typed C-zero     = ⊢zero
typed (C-suc CM) = ⊢suc (typed CM)
```

The proofs are straightforward, and again use induction in the case of successor.

Progress

We would like to show that every term is either a value or takes a reduction step. However, this is not true in general. The term

```
`zero · `suc `zero
```

is neither a value nor can take a reduction step. And if `s ⦂ `ℕ ⇒ `ℕ` then the term

```
 s · `zero
```

cannot reduce because we do not know which function is bound to the free variable `s`. The first of those terms is ill typed, and the second has a free variable. Every term that is well typed and closed has the desired property.

Progress: If `∅ ⊢ M ⦂ A` then either `M` is a value or there is an `N` such that `M ⟶ N`.

To formulate this property, we first introduce a relation that captures what it means for a term `M` to make progress:

```
data Progress (M : Term) : Set where

  step : ∀ {N}
    → M ⟶ N
      ----------
```

```
    → Progress M

  done :
     Value M
     ----------
    → Progress M
```

A term `M` makes progress if either it can take a step, meaning there exists a term `N` such that `M ⟶ N`, or if it is done, meaning that `M` is a value.

If a term is well typed in the empty context then it satisfies progress:

```
progress : ∀ {M A}
  → ∅ ⊢ M ⦂ A
    ----------
  → Progress M
progress (⊢` ())
progress (⊢ƛ ⊢N) = done V-ƛ
progress (⊢L · ⊢M) with progress ⊢L
... | step L⟶L′ = step (ξ-·₁ L⟶L′)
... | done VL with progress ⊢M
... | step M⟶M′ = step (ξ-·₂ VL M⟶M′)
... | done VM with canonical ⊢L VL
... | C-ƛ _       = step (β-ƛ VM)
progress ⊢zero   = done V-zero
progress (⊢suc ⊢M) with progress ⊢M
... | step M⟶M′ = step (ξ-suc M⟶M′)
... | done VM    = done (V-suc VM)
progress (⊢case ⊢L ⊢M ⊢N) with progress ⊢L
... | step L⟶L′ = step (ξ-case L⟶L′)
... | done VL with canonical ⊢L VL
... | C-zero     = step β-zero
... | C-suc CL   = step (β-suc (value CL))
progress (⊢μ ⊢M) = step β-μ
```

We induct on the evidence that the term is well typed. Let's unpack the first three cases:

- The term cannot be a variable, since no variable is well typed in the empty context.
- If the term is a lambda abstraction then it is a value.
- If the term is an application `L · M`, recursively apply progress to the derivation that `L` is well typed:
  - If the term steps, we have evidence that `L ⟶ L′`, which by `ξ-·₁` means that our original term steps to `L′ · M`
  - If the term is done, we have evidence that `L` is a value. Recursively apply progress to the derivation that `M` is well typed:
    * If the term steps, we have evidence that `M ⟶ M′`, which by `ξ-·₂` means that our original term steps to `L · M′`. Step `ξ-·₂` applies only if we have evidence that `L` is a value, but progress on that subterm has already supplied the required evidence.
    * If the term is done, we have evidence that `M` is a value. We apply the canonical forms lemma to the evidence that `L` is well typed and a value, which since we are in an application leads to the conclusion that `L` must be a lambda abstraction. We also have evidence that `M` is a value, so our original term steps by `β-ƛ`.

The remaining cases are similar. If by induction we have a `step` case we apply a `ξ` rule, and if we have a `done` case then either we have a value or apply a `β` rule. For fixpoint, no induction is required as the `β` rule applies immediately.

Our code reads neatly in part because we consider the `step` option before the `done` option. We could, of course, do it the other way around, but then the `...` abbreviation no longer works, and we will need to write out all the arguments in full. In general, the rule of thumb is to consider the easy case (here `step`) before the hard case (here `done`). If you have two hard cases, you will have to expand out `...` or introduce subsidiary functions.

Instead of defining a data type for `Progress M`, we could have formulated progress using disjunction and existentials:

```
postulate
  progress′ : ∀ M {A} → ∅ ⊢ M ⦂ A → Value M ⊎ ∃[ N ](M ⟶ N)
```

This leads to a less perspicuous proof. Instead of the mnemonic `done` and `step` we use `inj₁` and `inj₂`, and the term `N` is no longer implicit and so must be written out in full. In the case for `β-ƛ` this requires that we match against the lambda expression `L` to determine its bound variable and body, `ƛ x ⇒ N`, so we can show that `L · M` reduces to `N [ x := M ]`.

Exercise `Progress-≃` (practice)

Show that `Progress M` is isomorphic to `Value M ⊎ ∃[ N ](M ⟶ N)`.

```
-- Your code goes here
```

Exercise `progress′` (practice)

Write out the proof of `progress′` in full, and compare it to the proof of `progress` above.

```
-- Your code goes here
```

Exercise `value?` (practice)

Combine `progress` and `⟶¬V` to write a program that decides whether a well-typed term is a value:

```
postulate
  value? : ∀ {A M} → ∅ ⊢ M ⦂ A → Dec (Value M)
```

Prelude to preservation

The other property we wish to prove, preservation of typing under reduction, turns out to require considerably more work. The proof has three key steps.

The first step is to show that types are preserved by *renaming*.

Renaming: Let `Γ` and `Δ` be two contexts such that every variable that appears in `Γ` also appears with the same type in `Δ`. Then if any term is typeable under `Γ`, it has the same type under `Δ`.

In symbols:

```
∀ {x A} → Γ ∋ x ⦂ A  →  Δ ∋ x ⦂ A
-------------------------------
∀ {M A} → Γ ⊢ M ⦂ A  →  Δ ⊢ M ⦂ A
```

Three important corollaries follow. The *weaken* lemma asserts that a term which is well typed in the empty context is also well typed in an arbitrary context. The *drop* lemma asserts that a term which is well typed in a context where the same variable appears twice remains well typed if we drop the shadowed occurrence. The *swap* lemma asserts that a term which is well typed in a context remains well typed if we swap two variables.

(Renaming is similar to the *context invariance* lemma in *Software Foundations*, but it does not require the definition of `appears_free_in` nor the `free_in_context` lemma.)

The second step is to show that types are preserved by *substitution*.

Substitution: Say we have a closed term `V` of type `A`, and under the assumption that `x` has type `A` the term `N` has type `B`. Then substituting `V` for `x` in `N` yields a term that also has type `B`.

In symbols:

```
∅ ⊢ V ⦂ A
Γ , x ⦂ A ⊢ N ⦂ B
--------------------
Γ ⊢ N [ x := V ] ⦂ B
```

The result does not depend on `V` being a value, but it does require that `V` be closed; recall that we restricted our attention to substitution by closed terms in order to avoid the need to rename bound variables. The term into which we are substituting is typed in an arbitrary context `Γ`, extended by the variable `x` for which we are substituting; and the result term is typed in `Γ`.

The lemma establishes that substitution composes well with typing: typing the components separately guarantees that the result of combining them is also well typed.

The third step is to show preservation.

Preservation: If `∅ ⊢ M ⦂ A` and `M ⟶ N` then `∅ ⊢ N ⦂ A`.

The proof is by induction over the possible reductions, and the substitution lemma is crucial in showing that each of the `β` rules that uses substitution preserves types.

We now proceed with our three-step programme.

Renaming

We often need to "rebase" a type derivation, replacing a derivation `Γ ⊢ M ⦂ A` by a related derivation `Δ ⊢ M ⦂ A`. We may do so as long as every variable that appears in `Γ` also appears in `Δ`, and with the same type.

Three of the rules for typing (lambda abstraction, case on naturals, and fixpoint) have hypotheses that extend the context to include a bound variable. In each of these rules, `Γ` appears in the conclusion and `Γ , x ⦂ A` appears in a hypothesis. Thus:

```
Γ , x ⦂ A ⊢ N ⦂ B
------------------ ⊢ƛ
Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B
```

for lambda expressions, and similarly for case and fixpoint. To deal with this situation, we first prove a lemma showing that if one context maps to another, this is still true after adding the same variable to both contexts:

```
ext : ∀ {Γ Δ}
  → (∀ {x A} → Γ ∋ x ⦂ A → Δ ∋ x ⦂ A)
    ---------------------------------------------------
  → (∀ {x y A B} → Γ , y ⦂ B ∋ x ⦂ A → Δ , y ⦂ B ∋ x ⦂ A)
ext ρ Z          = Z
ext ρ (S x≠y ∋x) = S x≠y (ρ ∋x)
```

Let `ρ` be the name of the map that takes evidence that `x` appears in `Γ` to evidence that `x` appears in `Δ`. The proof is by case analysis of the evidence that `x` appears in the extended map `Γ , y ⦂ B`:

- If `x` is the same as `y`, we used `Z` to access the last variable in the extended `Γ`; and can similarly use `Z` to access the last variable in the extended `Δ`.
- If `x` differs from `y`, then we used `S` to skip over the last variable in the extended `Γ`, where `x≠y` is evidence that `x` and `y` differ, and `∋x` is the evidence that `x` appears in `Γ`; and we can similarly use `S` to skip over the last variable in the extended `Δ`, applying `ρ` to find the evidence that `x` appears in `Δ`.

With the extension lemma under our belts, it is straightforward to prove renaming preserves types:

```
rename : ∀ {Γ Δ}
  → (∀ {x A} → Γ ∋ x ⦂ A → Δ ∋ x ⦂ A)
    ----------------------------------
  → (∀ {M A} → Γ ⊢ M ⦂ A → Δ ⊢ M ⦂ A)
rename ρ (⊢` ∋w)           = ⊢` (ρ ∋w)
rename ρ (⊢ƛ ⊢N)           = ⊢ƛ (rename (ext ρ) ⊢N)
rename ρ (⊢L · ⊢M)         = (rename ρ ⊢L) · (rename ρ ⊢M)
rename ρ ⊢zero             = ⊢zero
rename ρ (⊢suc ⊢M)         = ⊢suc (rename ρ ⊢M)
rename ρ (⊢case ⊢L ⊢M ⊢N) = ⊢case (rename ρ ⊢L) (rename ρ ⊢M) (rename (ext ρ) ⊢N)
rename ρ (⊢μ ⊢M)           = ⊢μ (rename (ext ρ) ⊢M)
```

As before, let `ρ` be the name of the map that takes evidence that `x` appears in `Γ` to evidence that `x` appears in `Δ`. We induct on the evidence that `M` is well typed in `Γ`. Let's unpack the first three cases:

- If the term is a variable, then applying `ρ` to the evidence that the variable appears in `Γ` yields the corresponding evidence that the variable appears in `Δ`.
- If the term is a lambda abstraction, use the previous lemma to extend the map `ρ` suitably and use induction to rename the body of the abstraction.
- If the term is an application, use induction to rename both the function and the argument.

The remaining cases are similar, using induction for each subterm, and extending the map whenever the construct introduces a bound variable.

The induction is over the derivation that the term is well typed, so extending the context doesn't invalidate the inductive hypothesis. Equivalently, the recursion terminates because the second argument always grows smaller, even though the first argument sometimes grows larger.

We have three important corollaries, each proved by constructing a suitable map between contexts.

First, a closed term can be weakened to any context:

```
weaken : ∀ {Γ M A}
  → ∅ ⊢ M ⦂ A
    ----------
  → Γ ⊢ M ⦂ A
weaken {Γ} ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → ∅ ∋ z ⦂ C
      ---------
    → Γ ∋ z ⦂ C
  ρ ()
```

Here the map `ρ` is trivial, since there are no possible arguments in the empty context `∅`.

Second, if the last two variables in a context are equal then we can drop the shadowed one:

```
drop : ∀ {Γ x M A B C}
  → Γ , x ⦂ A , x ⦂ B ⊢ M ⦂ C
    -------------------------
  → Γ , x ⦂ B ⊢ M ⦂ C
drop {Γ} {x} {M} {A} {B} {C} ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → Γ , x ⦂ A , x ⦂ B ∋ z ⦂ C
      -------------------------
    → Γ , x ⦂ B ∋ z ⦂ C
  ρ Z                 = Z
  ρ (S x≠x Z)         = ⊥-elim (x≠x refl)
  ρ (S z≠x (S _ ∋z)) = S z≠x ∋z
```

Here map `ρ` can never be invoked on the inner occurrence of `x` since it is masked by the outer occurrence. Skipping over the `x` in the first position can only happen if the variable looked for differs from `x` (the evidence for which is `x≠x` or `z≠x`) but if the variable is found in the second position, which also contains `x`, this leads to a contradiction (evidenced by `x≠x refl`).

Third, if the last two variables in a context differ then we can swap them:

```
swap : ∀ {Γ x y M A B C}
  → x ≢ y
```

```
  → Γ , y ⦂ B , x ⦂ A ⊢ M ⦂ C
    ------------------------
  → Γ , x ⦂ A , y ⦂ B ⊢ M ⦂ C
swap {Γ} {x} {y} {M} {A} {B} {C} x≠y ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → Γ , y ⦂ B , x ⦂ A ∋ z ⦂ C
      ------------------------
    → Γ , x ⦂ A , y ⦂ B ∋ z ⦂ C
  ρ Z                 = S x≠y Z
  ρ (S z≠x Z)         = Z
  ρ (S z≠x (S z≠y ∋z)) = S z≠y (S z≠x ∋z)
```

Here the renaming map takes a variable at the end into a variable one from the end, and vice versa. The first line is responsible for moving `x` from a position at the end to a position one from the end with `y` at the end, and requires the provided evidence that `x ≠ y`.

Substitution

The key to preservation – and the trickiest bit of the proof – is the lemma establishing that substitution preserves types.

Recall that in order to avoid renaming bound variables, substitution is restricted to be by closed terms only. This restriction was not enforced by our definition of substitution, but it is captured by our lemma to assert that substitution preserves typing.

Our concern is with reducing closed terms, which means that when we apply `β` reduction, the term substituted in contains a single free variable (the bound variable of the lambda abstraction, or similarly for case or fixpoint). However, substitution is defined by recursion, and as we descend into terms with bound variables the context grows. So for the induction to go through, we require an arbitrary context `Γ`, as in the statement of the lemma.

Here is the formal statement and proof that substitution preserves types:

```
subst : ∀ {Γ x N V A B}
  → ∅ ⊢ V ⦂ A
  → Γ , x ⦂ A ⊢ N ⦂ B
    --------------------
  → Γ ⊢ N [ x := V ] ⦂ B
subst {x = y} ⊢V (⊢` {x = x} Z) with x ≟ y
... | yes _         = weaken ⊢V
... | no x≠y        = ⊥-elim (x≠y refl)
subst {x = y} ⊢V (⊢` {x = x} (S x≠y ∋x)) with x ≟ y
... | yes refl      = ⊥-elim (x≠y refl)
... | no _          = ⊢` ∋x
subst {x = y} ⊢V (⊢ƛ {x = x} ⊢N) with x ≟ y
... | yes refl      = ⊢ƛ (drop ⊢N)
... | no x≠y        = ⊢ƛ (subst ⊢V (swap x≠y ⊢N))
subst ⊢V (⊢L · ⊢M)  = (subst ⊢V ⊢L) · (subst ⊢V ⊢M)
subst ⊢V ⊢zero      = ⊢zero
subst ⊢V (⊢suc ⊢M)  = ⊢suc (subst ⊢V ⊢M)
subst {x = y} ⊢V (⊢case {x = x} ⊢L ⊢M ⊢N) with x ≟ y
... | yes refl      = ⊢case (subst ⊢V ⊢L) (subst ⊢V ⊢M) (drop ⊢N)
... | no x≠y        = ⊢case (subst ⊢V ⊢L) (subst ⊢V ⊢M) (subst ⊢V (swap x≠y ⊢N))
subst {x = y} ⊢V (⊢μ {x = x} ⊢M) with x ≟ y
```

```
... | yes refl     = ⊢μ (drop ⊢M)
... | no x≢y      = ⊢μ (subst ⊢V (swap x≢y ⊢M))
```

We induct on the evidence that `N` is well typed in the context `Γ` extended by `x`.

First, we note a wee issue with naming. In the lemma statement, the variable `x` is an implicit parameter for the variable substituted, while in the type rules for variables, abstractions, cases, and fixpoints, the variable `x` is an implicit parameter for the relevant variable. We are going to need to get hold of both variables, so we use the syntax `{x = y}` to bind `y` to the substituted variable and the syntax `{x = x}` to bind `x` to the relevant variable in the patterns for `⊢`` `, `⊢ƛ`, `⊢case`, and `⊢μ`. Using the name `y` here is consistent with the naming in the original definition of substitution in the previous chapter. The proof never mentions the types of `x`, `y`, `V`, or `N`, so in what follows we choose type names as convenient.

Now that naming is resolved, let's unpack the first three cases:

- In the variable case, we must show

  ```
  ∅ ⊢ V ⦂ B
  Γ , y ⦂ B ⊢ ` x ⦂ A
  ------------------------
  Γ ⊢ ` x [ y := V ] ⦂ A
  ```

  where the second hypothesis follows from:

  ```
  Γ , y ⦂ B ∋ x ⦂ A
  ```

  There are two subcases, depending on the evidence for this judgment:

  - The lookup judgment is evidenced by rule `Z`:

    ```
    ----------------
    Γ , x ⦂ A ∋ x ⦂ A
    ```

    In this case, `x` and `y` are necessarily identical, as are `A` and `B`. Nonetheless, we must evaluate `x ≟ y` in order to allow the definition of substitution to simplify:

    * If the variables are equal, then after simplification we must show

      ```
      ∅ ⊢ V ⦂ A
      ---------
      Γ ⊢ V ⦂ A
      ```

      which follows by weakening.
    * If the variables are unequal we have a contradiction.

  - The lookup judgment is evidenced by rule `S`:

    ```
    x ≢ y
    Γ ∋ x ⦂ A
    -----------------
    Γ , y ⦂ B ∋ x ⦂ A
    ```

    In this case, `x` and `y` are necessarily distinct. Nonetheless, we must again evaluate `x ≟ y` in order to allow the definition of substitution to simplify:

    * If the variables are equal we have a contradiction.
    * If the variables are unequal, then after simplification we must show

```
      ∅ ⊢ V ⦂ B
      x ≠ y
      Γ ∋ x ⦂ A
      -------------
      Γ ⊢ ` x ⦂ A
```

which follows by the typing rule for variables.

- In the abstraction case, we must show

```
∅ ⊢ V ⦂ B
Γ , y ⦂ B ⊢ (ƛ x ⇒ N) ⦂ A ⇒ C
--------------------------------
Γ ⊢ (ƛ x ⇒ N) [ y := V ] ⦂ A ⇒ C
```

where the second hypothesis follows from

```
Γ , y ⦂ B , x ⦂ A ⊢ N ⦂ C
```

We evaluate `x ≟ y` in order to allow the definition of substitution to simplify:

  - If the variables are equal then after simplification we must show:

```
    ∅ ⊢ V ⦂ B
    Γ , x ⦂ B , x ⦂ A ⊢ N ⦂ C
    ------------------------
    Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ C
```

    From the drop lemma, `drop`, we may conclude:

```
    Γ , x ⦂ B , x ⦂ A ⊢ N ⦂ C
    ------------------------
    Γ , x ⦂ A ⊢ N ⦂ C
```

    The typing rule for abstractions then yields the required conclusion.

  - If the variables are distinct then after simplification we must show:

```
    ∅ ⊢ V ⦂ B
    Γ , y ⦂ B , x ⦂ A ⊢ N ⦂ C
    --------------------------------
    Γ ⊢ ƛ x ⇒ (N [ y := V ]) ⦂ A ⇒ C
```

    From the swap lemma we may conclude:

```
    Γ , y ⦂ B , x ⦂ A ⊢ N ⦂ C
    ------------------------
    Γ , x ⦂ A , y ⦂ B ⊢ N ⦂ C
```

    The inductive hypothesis gives us:

```
    ∅ ⊢ V ⦂ B
    Γ , x ⦂ A , y ⦂ B ⊢ N ⦂ C
    ---------------------------
    Γ , x ⦂ A ⊢ N [ y := V ] ⦂ C
```

    The typing rule for abstractions then yields the required conclusion.

- In the application case, we must show

```
  ∅ ⊢ V ⦂ C
  Γ , y ⦂ C ⊢ L · M ⦂ B
  ------------------------
  Γ ⊢ (L · M) [ y := V ] ⦂ B
```

where the second hypothesis follows from the two judgments

```
  Γ , y ⦂ C ⊢ L ⦂ A ⇒ B
  Γ , y ⦂ C ⊢ M ⦂ A
```

By the definition of substitution, we must show:

```
  ∅ ⊢ V ⦂ C
  Γ , y ⦂ C ⊢ L ⦂ A ⇒ B
  Γ , y ⦂ C ⊢ M ⦂ A
  ------------------------------------
  Γ ⊢ (L [ y := V ]) · (M [ y := V ]) ⦂ B
```

Applying the induction hypothesis for `L` and `M` and the typing rule for applications yields the required conclusion.

The remaining cases are similar, using induction for each subterm. Where the construct introduces a bound variable we need to compare it with the substituted variable, applying the drop lemma if they are equal and the swap lemma if they are distinct.

For Agda it makes a difference whether we write `x ≟ y` or `y ≟ x`. In an interactive proof, Agda will show which residual `with` clauses in the definition of `_[_:=_]` need to be simplified, and the `with` clauses in `subst` need to match these exactly. The guideline is that Agda knows nothing about symmetry or commutativity, which require invoking appropriate lemmas, so it is important to think about order of arguments and to be consistent.

Exercise `subst′` (stretch)

Rewrite `subst` to work with the modified definition `_[_:=_]′` from the exercise in the previous chapter. As before, this should factor dealing with bound variables into a single function, defined by mutual recursion with the proof that substitution preserves types.

```
-- Your code goes here
```

Preservation

Once we have shown that substitution preserves types, showing that reduction preserves types is straightforward:

```
preserve : ∀ {M N A}
  → ∅ ⊢ M ⦂ A
  → M ⟶ N
    ----------
  → ∅ ⊢ N ⦂ A
preserve (⊢` ())
preserve (⊢ƛ ⊢N)           ()
```

```
preserve (⊢L · ⊢M)           (ξ-·₁ L⟶L′)    = (preserve ⊢L L⟶L′) · ⊢M
preserve (⊢L · ⊢M)           (ξ-·₂ VL M⟶M′) = ⊢L · (preserve ⊢M M⟶M′)
preserve ((⊢ƛ ⊢N) · ⊢V)      (β-ƛ VV)       = subst ⊢V ⊢N
preserve ⊢zero               ()
preserve (⊢suc ⊢M)           (ξ-suc M⟶M′)   = ⊢suc (preserve ⊢M M⟶M′)
preserve (⊢case ⊢L ⊢M ⊢N)    (ξ-case L⟶L′)  = ⊢case (preserve ⊢L L⟶L′) ⊢M ⊢N
preserve (⊢case ⊢zero ⊢M ⊢N) (β-zero)       = ⊢M
preserve (⊢case (⊢suc ⊢V) ⊢M ⊢N) (β-suc VV) = subst ⊢V ⊢N
preserve (⊢μ ⊢M)             (β-μ)          = subst (⊢μ ⊢M) ⊢M
```

The proof never mentions the types of `M` or `N`, so in what follows we choose type name as convenient.

Let's unpack the cases for two of the reduction rules:

- Rule `ξ-·₁`. We have

  ```
  L ⟶ L′
  ----------------
  L · M ⟶ L′ · M
  ```

  where the left-hand side is typed by

  ```
  Γ ⊢ L ⦂ A ⇒ B
  Γ ⊢ M ⦂ A
  -------------
  Γ ⊢ L · M ⦂ B
  ```

  By induction, we have

  ```
  Γ ⊢ L ⦂ A ⇒ B
  L ⟶ L′
  --------------
  Γ ⊢ L′ ⦂ A ⇒ B
  ```

  from which the typing of the right-hand side follows immediately.

- Rule `β-ƛ`. We have

  ```
  Value V
  ------------------------------
  (ƛ x ⇒ N) · V ⟶ N [ x := V ]
  ```

  where the left-hand side is typed by

  ```
  Γ , x ⦂ A ⊢ N ⦂ B
  -------------------
  Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B    Γ ⊢ V ⦂ A
  --------------------------------
  Γ ⊢ (ƛ x ⇒ N) · V ⦂ B
  ```

  By the substitution lemma, we have

  ```
  Γ ⊢ V ⦂ A
  Γ , x ⦂ A ⊢ N ⦂ B
  --------------------
  Γ ⊢ N [ x := V ] ⦂ B
  ```

  from which the typing of the right-hand side follows immediately.

The remaining cases are similar. Each `ξ` rule follows by induction, and each `β` rule follows by the substitution lemma.

Evaluation

By repeated application of progress and preservation, we can evaluate any well-typed term. In this section, we will present an Agda function that computes the reduction sequence from any given closed, well-typed term to its value, if it has one.

Some terms may reduce forever. Here is a simple example:

```
sucμ = μ "x" ⇒ `suc (` "x")

_ =
  begin
    sucμ
  ⟶⟨ β-μ ⟩
    `suc sucμ
  ⟶⟨ ξ-suc β-μ ⟩
    `suc `suc sucμ
  ⟶⟨ ξ-suc (ξ-suc β-μ) ⟩
    `suc `suc `suc sucμ
  -- ...
  ∎
```

Since every Agda computation must terminate, we cannot simply ask Agda to reduce a term to a value. Instead, we will provide a natural number to Agda, and permit it to stop short of a value if the term requires more than the given number of reduction steps.

A similar issue arises with cryptocurrencies. Systems which use smart contracts require the miners that maintain the blockchain to evaluate the program which embodies the contract. For instance, validating a transaction on Ethereum may require executing a program for the Ethereum Virtual Machine (EVM). A long-running or non-terminating program might cause the miner to invest arbitrary effort in validating a contract for little or no return. To avoid this situation, each transaction is accompanied by an amount of *gas* available for computation. Each step executed on the EVM is charged an advertised amount of gas, and the transaction pays for the gas at a published rate: a given number of Ethers (the currency of Ethereum) per unit of gas.

By analogy, we will use the name *gas* for the parameter which puts a bound on the number of reduction steps. `Gas` is specified by a natural number:

```
record Gas : Set where
  constructor gas
  field
    amount : ℕ
```

When our evaluator returns a term `N`, it will either give evidence that `N` is a value or indicate that it ran out of gas:

```
data Finished (N : Term) : Set where

  done :
    Value N
    ----------
    → Finished N

  out-of-gas :
```

```
    ----------
    Finished N
```

Given a term `L` of type `A`, the evaluator will, for some `N`, return a reduction sequence from `L` to `N` and an indication of whether reduction finished:

```
data Steps (L : Term) : Set where

  steps : ∀ {N}
    → L —↠ N
    → Finished N
      ----------
    → Steps L
```

The evaluator takes gas and evidence that a term is well typed, and returns the corresponding steps:

```
eval : ∀ {L A}
  → Gas
  → ∅ ⊢ L ⦂ A
    ---------
  → Steps L
eval {L} (gas zero) ⊢L = steps (L ∎) out-of-gas
eval {L} (gas (suc m)) ⊢L with progress ⊢L
... | done VL          = steps (L ∎) (done VL)
... | step {M} L—→M with eval (gas m) (preserve ⊢L L—→M)
... | steps M—↠N fin   = steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

Let `L` be the name of the term we are reducing, and `⊢L` be the evidence that `L` is well typed. We consider the amount of gas remaining. There are two possibilities:

- It is zero, so we stop early. We return the trivial reduction sequence `L —↠ L`, evidence that `L` is well typed, and an indication that we are out of gas.
- It is non-zero and after the next step we have `m` gas remaining. Apply progress to the evidence that term `L` is well typed. There are two possibilities:
  - Term `L` is a value, so we are done. We return the trivial reduction sequence `L —↠ L`, evidence that `L` is well typed, and the evidence that `L` is a value.
  - Term `L` steps to another term `M`. Preservation provides evidence that `M` is also well typed, and we recursively invoke `eval` on the remaining gas. The result is evidence that `M —↠ N`, together with evidence that `N` is well typed and an indication of whether reduction finished. We combine the evidence that `L —→ M` and `M —↠ N` to return evidence that `L —↠ N`, together with the other relevant evidence.

Examples

We can now use Agda to compute the non-terminating reduction sequence given earlier. First, we show that the term `sucμ` is well typed:

```
⊢sucμ : ∅ ⊢ μ "x" ⇒ `suc ` "x" ⦂ `ℕ
⊢sucμ = ⊢μ (⊢suc (⊢` ∋x))
  where
  ∋x = Z
```

To show the first three steps of the infinite reduction sequence, we evaluate with three steps worth of gas:

```
_ : eval (gas 3) ⊢sucμ ≡
  steps
    (μ "x" ⇒ `suc ` "x"
    ⟶⟨ β-μ ⟩
      `suc (μ "x" ⇒ `suc ` "x")
    ⟶⟨ ξ-suc β-μ ⟩
      `suc (`suc (μ "x" ⇒ `suc ` "x"))
    ⟶⟨ ξ-suc (ξ-suc β-μ) ⟩
      `suc (`suc (`suc (μ "x" ⇒ `suc ` "x")))
    ∎)
    out-of-gas
_ = refl
```

Similarly, we can use Agda to compute the reduction sequences given in the previous chapter. We start with the Church numeral two applied to successor and zero. Supplying 100 steps of gas is more than enough:

```
_ : eval (gas 100) (⊢two꜀ · ⊢suc꜀ · ⊢zero) ≡
  steps
    ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
    · `zero
    ⟶⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
      (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
        `zero
    ⟶⟨ β-ƛ V-zero ⟩
      (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · `zero)
    ⟶⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
      (ƛ "n" ⇒ `suc ` "n") · `suc `zero
    ⟶⟨ β-ƛ (V-suc V-zero) ⟩
      `suc (`suc `zero)
    ∎)
    (done (V-suc (V-suc V-zero)))
_ = refl
```

The example above was generated by using `C-c C-n` to normalise the left-hand side of the equation and pasting in the result as the right-hand side of the equation. The example reduction of the previous chapter was derived from this result, reformatting and writing `two꜀` and `suc꜀` in place of their expansions.

Next, we show two plus two is four:

```
_ : eval (gas 100) ⊢2+2 ≡
  steps
    ((μ "+" ⇒
       (ƛ "m" ⇒
         (ƛ "n" ⇒
           case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
           ])))
      · `suc (`suc `zero)
      · `suc (`suc `zero)
    ⟶⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
      (ƛ "m" ⇒
        (ƛ "n" ⇒
          case ` "m" [zero⇒ ` "n" |suc "m" ⇒
          `suc
```

```
      ((μ "+" ⇒
         (ƛ "m" ⇒
           (ƛ "n" ⇒
             case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
             ])))
        · ` "m"
        · ` "n")
      ]))
  · `suc (`suc `zero)
  · `suc (`suc `zero)
⟶⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
  (ƛ "n" ⇒
   case `suc (`suc `zero) [zero⇒ ` "n" |suc "m" ⇒
   `suc
   ((μ "+" ⇒
      (ƛ "m" ⇒
        (ƛ "n" ⇒
         case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
         ])))
     · ` "m"
     · ` "n")
   ])
  · `suc (`suc `zero)
⟶⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
  case `suc (`suc `zero) [zero⇒ `suc (`suc `zero) |suc "m" ⇒
  `suc
  ((μ "+" ⇒
     (ƛ "m" ⇒
       (ƛ "n" ⇒
        case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
        ])))
    · ` "m"
    · `suc (`suc `zero))
  ]
⟶⟨ β-suc (V-suc V-zero) ⟩
  `suc
  ((μ "+" ⇒
     (ƛ "m" ⇒
       (ƛ "n" ⇒
        case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
        ])))
    · `suc `zero
    · `suc (`suc `zero))
⟶⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
  `suc
  ((ƛ "m" ⇒
     (ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒
      `suc
      ((μ "+" ⇒
         (ƛ "m" ⇒
           (ƛ "n" ⇒
             case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
             ])))
        · ` "m"
        · ` "n")
      ]))
    · `suc `zero
    · `suc (`suc `zero))
```

```
    ⟶⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
      `suc
      ((ƛ "n" ⇒
         case `suc `zero [zero⇒ ` "n" |suc "m" ⇒
         `suc
         ((μ "+" ⇒
            (ƛ "m" ⇒
              (ƛ "n" ⇒
                case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
                ])))
           · ` "m"
           · ` "n")
         ])
        · `suc (`suc `zero))
    ⟶⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
      `suc
      case `suc `zero [zero⇒ `suc (`suc `zero) |suc "m" ⇒
      `suc
      ((μ "+" ⇒
         (ƛ "m" ⇒
           (ƛ "n" ⇒
             case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
             ])))
        · ` "m"
        · `suc (`suc `zero))
      ]
    ⟶⟨ ξ-suc (β-suc V-zero) ⟩
      `suc
      (`suc
        ((μ "+" ⇒
           (ƛ "m" ⇒
             (ƛ "n" ⇒
               case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
               ])))
          · `zero
          · `suc (`suc `zero)))
    ⟶⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
      `suc
      (`suc
        ((ƛ "m" ⇒
           (ƛ "n" ⇒
             case ` "m" [zero⇒ ` "n" |suc "m" ⇒
             `suc
             ((μ "+" ⇒
                (ƛ "m" ⇒
                  (ƛ "n" ⇒
                    case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
                    ])))
               · ` "m"
               · ` "n")
             ]))
          · `zero
          · `suc (`suc `zero)))
    ⟶⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
      `suc
      (`suc
        ((ƛ "n" ⇒
           case `zero [zero⇒ ` "n" |suc "m" ⇒
           `suc
```

```
          ((μ "+" ⇒
             (ƛ "m" ⇒
               (ƛ "n" ⇒
                 case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
                 ])))
            · ` "m"
            · ` "n")
          ])
        · `suc (`suc `zero)))
    ⟶⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
      `suc
      (`suc
       case `zero [zero⇒ `suc (`suc `zero) |suc "m" ⇒
       `suc
       ((μ "+" ⇒
          (ƛ "m" ⇒
            (ƛ "n" ⇒
              case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
              ])))
         · ` "m"
         · `suc (`suc `zero))
       ])
    ⟶⟨ ξ-suc (ξ-suc β-zero) ⟩
      `suc (`suc (`suc (`suc `zero)))
    ∎)
    (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

Again, the derivation in the previous chapter was derived by editing the above.

Similarly, we can evaluate the corresponding term for Church numerals:

```
_ : eval (gas 100) ⊢2+2ᶜ ≡
  steps
    ((ƛ "m" ⇒
       (ƛ "n" ⇒
         (ƛ "s" ⇒ (ƛ "z" ⇒ ` "m" · ` "s" · (` "n" · ` "s" · ` "z")))))
      · (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")))
      · (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")))
      · (ƛ "n" ⇒ `suc ` "n")
      · `zero
    ⟶⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
      (ƛ "n" ⇒
        (ƛ "s" ⇒
          (ƛ "z" ⇒
            (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · ` "s" ·
            (` "n" · ` "s" · ` "z"))))
      · (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")))
      · (ƛ "n" ⇒ `suc ` "n")
      · `zero
    ⟶⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
      (ƛ "s" ⇒
        (ƛ "z" ⇒
          (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · ` "s" ·
          ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · ` "s" · ` "z")))
      · (ƛ "n" ⇒ `suc ` "n")
      · `zero
    ⟶⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
      (ƛ "z" ⇒
```

```
      (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
      ·
      ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
        · ` "z"))
    · `zero
  ⟶⟨ β-ƛ V-zero ⟩
    (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
    ·
    ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
      · `zero)
  ⟶⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
    ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
      · `zero)
  ⟶⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
    ((ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
      `zero)
  ⟶⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
    ((ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · `zero))
  ⟶⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
    (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
    ((ƛ "n" ⇒ `suc ` "n") · `suc `zero)
  ⟶⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
    (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
    `suc (`suc `zero)
  ⟶⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · `suc (`suc `zero))
  ⟶⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ "n" ⇒ `suc ` "n") · `suc (`suc (`suc `zero))
  ⟶⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎)
  (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

And again, the example in the previous section was derived by editing the above.

Exercise `mul-eval` (recommended)

Using the evaluator, confirm that two times two is four.

```
-- Your code goes here
```

Exercise: `progress-preservation` (practice)

Without peeking at their statements above, write down the progress and preservation theorems for the simply typed lambda-calculus.

```
-- Your code goes here
```

Exercise `subject_expansion` **(practice)**

We say that `M` *reduces* to `N` if `M ⟶ N`, but we can also describe the same situation by saying that `N` *expands* to `M`. The preservation property is sometimes called *subject reduction*. Its opposite is *subject expansion*, which holds if `M ⟶ N` and `∅ ⊢ N ⦂ A` imply `∅ ⊢ M ⦂ A`. Find two counter-examples to subject expansion, one with case expressions and one not involving case expressions.

```
-- Your code goes here
```

Well-typed terms don't get stuck

A term is *normal* if it cannot reduce:

```
Normal : Term → Set
Normal M = ∀ {N} → ¬ (M ⟶ N)
```

A term is *stuck* if it is normal yet not a value:

```
Stuck : Term → Set
Stuck M = Normal M × ¬ Value M
```

Using progress, it is easy to show that no well-typed term is stuck:

```
postulate
  unstuck : ∀ {M A}
    → ∅ ⊢ M ⦂ A
      -----------
    → ¬ (Stuck M)
```

Using preservation, it is easy to show that after any number of steps, a well-typed term remains well typed:

```
postulate
  preserves : ∀ {M N A}
    → ∅ ⊢ M ⦂ A
    → M ⟶↠ N
      ---------
    → ∅ ⊢ N ⦂ A
```

An easy consequence is that starting from a well-typed term, taking any number of reduction steps leads to a term that is not stuck:

```
postulate
  wttdgs : ∀ {M N A}
    → ∅ ⊢ M ⦂ A
    → M ⟶↠ N
      -----------
    → ¬ (Stuck N)
```

Felleisen and Wright, who introduced proofs via progress and preservation, summarised this result with the slogan *well-typed terms don't get stuck*. (They were referring to earlier work by Robin

Milner, who used denotational rather than operational semantics. He introduced `wrong` as the denotation of a term with a type error, and showed *well-typed terms don't go wrong*.)

Exercise `stuck` (practice)

Give an example of an ill-typed term that does get stuck.

```
-- Your code goes here
```

Exercise `unstuck` (recommended)

Provide proofs of the three postulates, `unstuck`, `preserves`, and `wttdgs` above.

```
-- Your code goes here
```

Reduction is deterministic

When we introduced reduction, we claimed it was deterministic. For completeness, we present a formal proof here.

Our proof will need a variant of congruence to deal with functions of four arguments (to deal with `case_[zero⇒_|suc_⇒_]`). It is exactly analogous to `cong` and `cong₂` as defined previously:

```
cong₄ : ∀ {A B C D E : Set} (f : A → B → C → D → E)
  {s w : A} {t x : B} {u y : C} {v z : D}
  → s ≡ w → t ≡ x → u ≡ y → v ≡ z → f s t u v ≡ f w x y z
cong₄ f refl refl refl refl = refl
```

It is now straightforward to show that reduction is deterministic:

```
det : ∀ {M M′ M″}
  → (M —→ M′)
  → (M —→ M″)
    --------
  → M′ ≡ M″
det (ξ-·₁ L—→L′)  (ξ-·₁ L—→L″)    = cong₂ _·_ (det L—→L′ L—→L″) refl
det (ξ-·₁ L—→L′)  (ξ-·₂ VL M—→M″) = ⊥-elim (V¬—→ VL L—→L′)
det (ξ-·₁ L—→L′)  (β-ƛ _)         = ⊥-elim (V¬—→ V-ƛ L—→L′)
det (ξ-·₂ VL _)   (ξ-·₁ L—→L″)    = ⊥-elim (V¬—→ VL L—→L″)
det (ξ-·₂ _ M—→M′) (ξ-·₂ _ M—→M″) = cong₂ _·_ refl (det M—→M′ M—→M″)
det (ξ-·₂ _ M—→M′) (β-ƛ VM)       = ⊥-elim (V¬—→ VM M—→M′)
det (β-ƛ _)       (ξ-·₁ L—→L″)    = ⊥-elim (V¬—→ V-ƛ L—→L″)
det (β-ƛ VM)      (ξ-·₂ _ M—→M″)  = ⊥-elim (V¬—→ VM M—→M″)
det (β-ƛ _)       (β-ƛ _)         = refl
det (ξ-suc M—→M′) (ξ-suc M—→M″)   = cong `suc_ (det M—→M′ M—→M″)
det (ξ-case L—→L′) (ξ-case L—→L″) = cong₄ case_[zero⇒_|suc_⇒_]
                                    (det L—→L′ L—→L″) refl refl refl
det (ξ-case L—→L′) β-zero         = ⊥-elim (V¬—→ V-zero L—→L′)
det (ξ-case L—→L′) (β-suc VL)     = ⊥-elim (V¬—→ (V-suc VL) L—→L′)
det β-zero        (ξ-case M—→M″)  = ⊥-elim (V¬—→ V-zero M—→M″)
```

```
det β-zero       β-zero         = refl
det (β-suc VL)   (ξ-case L⟶L″)  = ⊥-elim (V¬⟶ (V-suc VL) L⟶L″)
det (β-suc _)    (β-suc _)      = refl
det β-μ          β-μ            = refl
```

The proof is by induction over possible reductions. We consider three typical cases:

- Two instances of `ξ-·₁`:

  ```
  L ⟶ L′                  L ⟶ L″
  --------------- ξ-·₁    --------------- ξ-·₁
  L · M ⟶ L′ · M          L · M ⟶ L″ · M
  ```

  By induction we have `L′ ≡ L″`, and hence by congruence `L′ · M ≡ L″ · M`.

- An instance of `ξ-·₁` and an instance of `ξ-·₂`:

  ```
                          Value L
  L ⟶ L′                  M ⟶ M″
  --------------- ξ-·₁    --------------- ξ-·₂
  L · M ⟶ L′ · M          L · M ⟶ L · M″
  ```

  The rule on the left requires `L` to reduce, but the rule on the right requires `L` to be a value. This is a contradiction since values do not reduce. If the value constraint was removed from `ξ-·₂`, or from one of the other reduction rules, then determinism would no longer hold.

- Two instances of `β-ƛ`:

  ```
  Value V                              Value V
  ------------------------------ β-ƛ    ------------------------------ β-ƛ
  (ƛ x ⇒ N) · V ⟶ N [ x := V ]         (ƛ x ⇒ N) · V ⟶ N [ x := V ]
  ```

  Since the left-hand sides are identical, the right-hand sides are also identical. The formal proof simply invokes `refl`.

Five of the 18 lines in the above proof are redundant, e.g., the case when one rule is `ξ-·₁` and the other is `ξ-·₂` is considered twice, once with `ξ-·₁` first and `ξ-·₂` second, and the other time with the two swapped. What we might like to do is delete the redundant lines and add

```
det M⟶M′ M⟶M″ = sym (det M⟶M″ M⟶M′)
```

to the bottom of the proof. But this does not work: the termination checker complains, because the arguments have merely switched order and neither is smaller.

Quiz

Suppose we add a new term `zap` with the following reduction rule

```
-------- β-zap
M ⟶ zap
```

and the following typing rule:

```
----------- ⊢zap
Γ ⊢ zap ⦂ A
```

Which of the following properties remain true in the presence of these rules? For each property, write either "remains true" or "becomes false." If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Quiz

Suppose instead that we add a new term `foo` with the following reduction rules:

```
------------------ β-foo₁
(λ x ⇒ ` x) ⟶ foo

----------- β-foo₂
foo ⟶ zero
```

Which of the following properties remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Quiz

Suppose instead that we remove the rule `ξ·₁` from the step relation. Which of the following properties remain true in the absence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Quiz

We can enumerate all the computable function from naturals to naturals, by writing out all programs of type `` `ℕ ⇒ `ℕ `` in lexical order. Write `fᵢ` for the `i`'th function in this list.

Say we add a typing rule that applies the above enumeration to interpret a natural as a function from naturals to naturals:

```
Γ ⊢ L ⦂ `ℕ
Γ ⊢ M ⦂ `ℕ
-------------- _·ℕ_
Γ ⊢ L · M ⦂ `ℕ
```

And that we add the corresponding reduction rule:

```
fᵢ(m) ⟶ n
---------- δ
i · m ⟶ n
```

Which of the following properties remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Are all properties preserved in this case? Are there any other alterations we would wish to make to the system?

Unicode

This chapter uses the following unicode:

```
ƛ  U+019B  LATIN SMALL LETTER LAMBDA WITH STROKE (\Gl-)
Δ  U+0394  GREEK CAPITAL LETTER DELTA (\GD or \Delta)
β  U+03B2  GREEK SMALL LETTER BETA (\Gb or \beta)
δ  U+03B4  GREEK SMALL LETTER DELTA (\Gd or \delta)
μ  U+03BC  GREEK SMALL LETTER MU (\Gm or \mu)
ξ  U+03BE  GREEK SMALL LETTER XI (\Gx or \xi)
ρ  U+03B4  GREEK SMALL LETTER RHO (\Gr or \rho)
ᵢ  U+1D62  LATIN SUBSCRIPT SMALL LETTER I (\_i)
ᶜ  U+1D9C  MODIFIER LETTER SMALL C (\^c)
—  U+2013  EM DASH (\em)
₄  U+2084  SUBSCRIPT FOUR (\_4)
↠  U+21A0  RIGHTWARDS TWO HEADED ARROW (\rr-)
⇒  U+21D2  RIGHTWARDS DOUBLE ARROW (\=>)
∅  U+2205  EMPTY SET (\0)
∋  U+220B  CONTAINS AS MEMBER (\ni)
≟  U+225F  QUESTIONED EQUAL TO (\?=)
⊢  U+22A2  RIGHT TACK (\vdash or \|-)
⦂  U+2982  Z NOTATION TYPE COLON (\:)
```

Chapter 13

DeBruijn: Intrinsically-typed de Bruijn representation

```
module plfa.part2.DeBruijn where
```

The previous two chapters introduced lambda calculus, with a formalisation based on named variables, and terms defined separately from types. We began with that approach because it is traditional, but it is not the one we recommend. This chapter presents an alternative approach, where named variables are replaced by de Bruijn indices and terms are indexed by their types. Our new presentation is more compact, using substantially fewer lines of code to cover the same ground.

There are two fundamental approaches to typed lambda calculi. One approach, followed in the last two chapters, is to first define terms and then define types. Terms exist independent of types, and may have types assigned to them by separate typing rules. Another approach, followed in this chapter, is to first define types and then define terms. Terms and type rules are intertwined, and it makes no sense to talk of a term without a type. The two approaches are sometimes called *Curry style* and *Church style*. Following Reynolds, we will refer to them as *extrinsic* and *intrinsic*.

The particular representation described here was first proposed by Thorsten Altenkirch and Bernhard Reus. The formalisation of renaming and substitution we use is due to Conor McBride. Related work has been carried out by James Chapman, James McKinna, and many others.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _<_; _≤?_; z≤n; s≤s)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Decidable using (True; toWitness)
```

Introduction

There is a close correspondence between the structure of a term and the structure of the derivation showing that it is well typed. For example, here is the term for the Church numeral two:

```
twoᶜ : Term
twoᶜ = ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")
```

And here is its corresponding type derivation:

```
⊢twoᶜ : ∀ {A} → ∅ ⊢ twoᶜ ⦂ Ch A
⊢twoᶜ = ⊢ƛ (⊢ƛ (⊢` ∋s · (⊢` ∋s · ⊢` ∋z)))
  where
  ∋s = S ("s" ≠ "z") Z
  ∋z = Z
```

(These are both taken from Chapter Lambda and you can see the corresponding derivation tree written out in full here.) The two definitions are in close correspondence, where:

- `` `_ `` corresponds to `` ⊢` ``
- `` ƛ_⇒_ `` corresponds to `` ⊢ƛ ``
- `` _·_ `` corresponds to `` _·_ ``

Further, if we think of `Z` as zero and `S` as successor, then the lookup derivation for each variable corresponds to a number which tells us how many enclosing binding terms to count to find the binding of that variable. Here `"z"` corresponds to `Z` or zero and `"s"` corresponds to `S Z` or one. And, indeed, `"z"` is bound by the inner abstraction (count outward past zero abstractions) and `"s"` is bound by the outer abstraction (count outward past one abstraction).

In this chapter, we are going to exploit this correspondence, and introduce a new notation for terms that simultaneously represents the term and its type derivation. Now we will write the following:

```
twoᶜ  :  ∅ ⊢ Ch `ℕ
twoᶜ  =  ƛ ƛ (# 1 · (# 1 · # 0))
```

A variable is represented by a natural number (written with `Z` and `S`, and abbreviated in the usual way), and tells us how many enclosing binding terms to count to find the binding of that variable. Thus, `# 0` is bound at the inner `ƛ`, and `# 1` at the outer `ƛ`.

Replacing variables by numbers in this way is called *de Bruijn representation*, and the numbers themselves are called *de Bruijn indices*, after the Dutch mathematician Nicolaas Govert (Dick) de Bruijn (1918—2012), a pioneer in the creation of proof assistants. One advantage of replacing named variables with de Bruijn indices is that each term now has a unique representation, rather than being represented by the equivalence class of terms under alpha renaming.

The other important feature of our chosen representation is that it is *intrinsically typed*. In the previous two chapters, the definition of terms and the definition of types are completely separate. All terms have type `Term`, and nothing in Agda prevents one from writing a nonsense term such as `` `zero · `suc `zero `` which has no type. Such terms that exist independent of types are sometimes called *preterms* or *raw terms*. Here we are going to replace the type `Term` of raw terms by the type `Γ ⊢ A` of intrinsically-typed terms which in context `Γ` have type `A`.

While these two choices fit well, they are independent. One can use de Bruijn indices in raw terms, or have intrinsically-typed terms with names. In Chapter Untyped, we will introduce terms with de Bruijn indices that are intrinsically scoped but not typed.

A second example

De Bruijn indices can be tricky to get the hang of, so before proceeding further let's consider a second example. Here is the term that adds two naturals:

```
plus : Term
plus = μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
         case ` "m"
           [zero⇒ ` "n"
           |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

Note variable `"m"` is bound twice, once in a lambda abstraction and once in the successor branch of the case. Any appearance of `"m"` in the successor branch must refer to the latter binding, due to shadowing.

Here is its corresponding type derivation:

```
⊢plus : ∅ ⊢ plus ⦂ `ℕ ⇒ `ℕ ⇒ `ℕ
⊢plus = ⊢μ (⊢ƛ (⊢ƛ (⊢case (⊢` ∋m) (⊢` ∋n)
          (⊢suc (⊢` ∋+ · ⊢` ∋m′ · ⊢` ∋n′)))))
  where
  ∋+  = (S ("+" ≠ "m") (S ("+" ≠ "n") (S ("+" ≠ "m") Z)))
  ∋m  = (S ("m" ≠ "n") Z)
  ∋n  = Z
  ∋m′ = Z
  ∋n′ = (S ("n" ≠ "m") Z)
```

The two definitions are in close correspondence, where in addition to the previous correspondences we have:

- `` `zero `` corresponds to `⊢zero`
- `` `suc_ `` corresponds to `⊢suc`
- `case_[zero⇒_|suc_⇒_]` corresponds to `⊢case`
- `μ_⇒_` corresponds to `⊢μ`

Note the two lookup judgments `∋m` and `∋m′` refer to two different bindings of variables named `"m"`. In contrast, the two judgments `∋n` and `∋n′` both refer to the same binding of `"n"` but accessed in different contexts, the first where `"n"` is the last binding in the context, and the second after `"m"` is bound in the successor branch of the case.

Here is the term and its type derivation in the notation of this chapter:

```
plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = μ ƛ ƛ case (# 1) (# 0) (`suc (# 3 · # 0 · # 1))
```

Reading from left to right, each de Bruijn index corresponds to a lookup derivation:

- `# 1` corresponds to `∋m`
- `# 0` corresponds to `∋n`
- `# 3` corresponds to `∋+`
- `# 0` corresponds to `∋m′`
- `# 1` corresponds to `∋n′`

The de Bruijn index counts the number of `S` constructs in the corresponding lookup derivation. Variable `"n"` bound in the inner abstraction is referred to as `# 0` in the zero branch of the case

but as `# 1` in the successor branch of the case, because of the intervening binding. Variable `"m"` bound in the lambda abstraction is referred to by the first `# 1` in the code, while variable `"m"` bound in the successor branch of the case is referred to by the second `# 0`. There is no shadowing: with variable names, there is no way to refer to the former binding in the scope of the latter, but with de Bruijn indices it could be referred to as `# 2`.

Order of presentation

In the current chapter, the use of intrinsically-typed terms necessitates that we cannot introduce operations such as substitution or reduction without also showing that they preserve types. Hence, the order of presentation must change.

The syntax of terms now incorporates their typing rules, and the definition of values now incorporates the Canonical Forms lemma. The definition of substitution is somewhat more involved, but incorporates the trickiest part of the previous proof, the lemma establishing that substitution preserves types. The definition of reduction incorporates preservation, which no longer requires a separate proof.

Syntax

We now begin our formal development.

First, we get all our infix declarations out of the way. We list separately operators for judgments, types, and terms:

```
infix 4 _⊢_
infix 4 _∋_
infixl 5 _,_

infixr 7 _⇒_

infix 5 ƛ_
infix 5 μ_
infixl 7 _·_
infix 8 `suc_
infix 9 `_
infix 9 S_
infix 9 #_
```

Since terms are intrinsically typed, we must define types and contexts before terms.

Types

As before, we have just two types, functions and naturals. The formal definition is unchanged:

```
data Type : Set where
  _⇒_ : Type → Type → Type
  `ℕ : Type
```

Contexts

Contexts are as before, but we drop the names. Contexts are formalised as follows:

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

A context is just a list of types, with the type of the most recently bound variable on the right. As before, we let `Γ` and `Δ` range over contexts. We write `∅` for the empty context, and `Γ , A` for the context `Γ` extended by type `A`. For example

```
_ : Context
_ = ∅ , `ℕ ⇒ `ℕ , `ℕ
```

is a context with two variables in scope, where the outer bound one has type `` `ℕ ⇒ `ℕ ``, and the inner bound one has type `` `ℕ ``.

Variables and the lookup judgment

Intrinsically-typed variables correspond to the lookup judgment. They are represented by de Bruijn indices, and hence also correspond to natural numbers. We write

```
Γ ∋ A
```

for variables which in context `Γ` have type `A`. The lookup judgement is formalised by a datatype indexed by a context and a type. It looks exactly like the old lookup judgment, but with all variable names dropped:

```
data _∋_ : Context → Type → Set where

  Z : ∀ {Γ A}
    ---------
    → Γ , A ∋ A

  S_ : ∀ {Γ A B}
    → Γ ∋ A
      ---------
    → Γ , B ∋ A
```

Constructor `S` no longer requires an additional parameter, since without names shadowing is no longer an issue. Now constructors `Z` and `S` correspond even more closely to the constructors `here` and `there` for the element-of relation `_∈_` on lists, as well as to constructors `zero` and `suc` for natural numbers.

For example, consider the following old-style lookup judgments:

- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "s" ⦂ `ℕ ⇒ `ℕ`

They correspond to the following intrinsically-typed variables:

```
_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ∋ `ℕ
_ = Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ∋ `ℕ ⇒ `ℕ
_ = S Z
```

In the given context, `"z"` is represented by `Z` (as the most recently bound variable), and `"s"` by `S Z` (as the next most recently bound variable).

Terms and the typing judgment

Intrinsically-typed terms correspond to the typing judgment. We write

```
Γ ⊢ A
```

for terms which in context `Γ` have type `A`. The judgement is formalised by a datatype indexed by a context and a type. It looks exactly like the old typing judgment, but with all terms and variable names dropped:

```
data _⊢_ : Context → Type → Set where

  `_ : ∀ {Γ A}
    → Γ ∋ A
      -----
    → Γ ⊢ A

  ƛ_ : ∀ {Γ A B}
    → Γ , A ⊢ B
      ---------
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
      ---------
    → Γ ⊢ B

  `zero : ∀ {Γ}
      ---------
    → Γ ⊢ `ℕ

  `suc_ : ∀ {Γ}
    → Γ ⊢ `ℕ
      ------
    → Γ ⊢ `ℕ

  case : ∀ {Γ A}
    → Γ ⊢ `ℕ
    → Γ ⊢ A
    → Γ , `ℕ ⊢ A
      ----------
    → Γ ⊢ A

  μ_ : ∀ {Γ A}
    → Γ , A ⊢ A
      ---------
    → Γ ⊢ A
```

The definition exploits the close correspondence between the structure of terms and the structure of a derivation showing that it is well typed: now we use the derivation *as* the term.

For example, consider the following old-style typing judgments:

- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" ⦂ `ℕ ⇒ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" · ` "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" · (` "s" · ` "z") ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ ⊢ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) ⦂ `ℕ ⇒ `ℕ`
- `∅ ⊢ ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) ⦂ (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ`

They correspond to the following intrinsically-typed terms:

```
_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ
_ = ` Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ ⇒ `ℕ
_ = ` S Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ
_ = ` S Z · ` Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ
_ = ` S Z · (` S Z · ` Z)

_ : ∅ , `ℕ ⇒ `ℕ ⊢ `ℕ ⇒ `ℕ
_ = ƛ (` S Z · (` S Z · ` Z))

_ : ∅ ⊢ (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ
_ = ƛ ƛ (` S Z · (` S Z · ` Z))
```

The final term represents the Church numeral two.

Abbreviating de Bruijn indices

We define a helper function that computes the length of a context, which will be useful in making sure an index is within context bounds:

```
length : Context → ℕ
length ∅       = zero
length (Γ , _) = suc (length Γ)
```

We can use a natural number to select a type from a context:

```
lookup : {Γ : Context} → {n : ℕ} → (p : n < length Γ) → Type
lookup {(_ , A)} {zero} (s≤s z≤n) = A
lookup {(Γ , _)} {(suc n)} (s≤s p) = lookup p
```

We intend to apply the function only when the natural is shorter than the length of the context, which is witnessed by `p`.

Given the above, we can convert a natural to a corresponding de Bruijn index, looking up its type in the context:

```
count : ∀ {Γ} → {n : ℕ} → (p : n < length Γ) → Γ ∋ lookup p
count {_ , _} {zero} (s≤s z≤n) = Z
count {Γ , _} {(suc n)} (s≤s p) = S (count p)
```

We can then introduce a convenient abbreviation for variables:

```
#_ : ∀ {Γ}
  → (n : ℕ)
  → {n∈Γ : True (suc n ≤? length Γ)}
    --------------------------------
  → Γ ⊢ lookup (toWitness n∈Γ)
#_ n {n∈Γ} = ` count (toWitness n∈Γ)
```

Function `#_` takes an implicit argument `n∈Γ` that provides evidence for `n` to be within the context's bounds. Recall that `True`, `_≤?_` and `toWitness` are defined in Chapter Decidable. The type of `n∈Γ` guards against invoking `#_` on an `n` that is out of context bounds. Finally, in the return type `n∈Γ` is converted to a witness that `n` is within the bounds.

With this abbreviation, we can rewrite the Church numeral two more compactly:

```
_ : ∅ ⊢ (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ
_ = ƛ ƛ (# 1 · (# 1 · # 0))
```

Test examples

We repeat the test examples from Chapter Lambda. You can find them here for comparison.

First, computing two plus two on naturals:

```
two : ∀ {Γ} → Γ ⊢ `ℕ
two = `suc `suc `zero

plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = μ ƛ ƛ (case (# 1) (# 0) (`suc (# 3 · # 0 · # 1)))

2+2 : ∀ {Γ} → Γ ⊢ `ℕ
2+2 = plus · two · two
```

We generalise to arbitrary contexts because later we will give examples where `two` appears nested inside binders.

Next, computing two plus two on Church numerals:

```
Ch : Type → Type
Ch A = (A ⇒ A) ⇒ A ⇒ A

two^c : ∀ {Γ A} → Γ ⊢ Ch A
two^c = ƛ ƛ (# 1 · (# 1 · # 0))

plus^c : ∀ {Γ A} → Γ ⊢ Ch A ⇒ Ch A ⇒ Ch A
plus^c = ƛ ƛ ƛ ƛ (# 3 · # 1 · (# 2 · # 1 · # 0))

suc^c : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ
suc^c = ƛ `suc (# 0)
```

```
2+2ᶜ : ∀ {Γ} → Γ ⊢ `ℕ
2+2ᶜ = plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

As before we generalise everything to arbitrary contexts. While we are at it, we also generalise `twoᶜ` and `plusᶜ` to Church numerals over arbitrary types.

Exercise `mul` (recommended)

Write out the definition of a lambda term that multiplies two natural numbers, now adapted to the intrinsically-typed DeBruijn representation.

```
-- Your code goes here
```

Renaming

Renaming is a necessary prelude to substitution, enabling us to "rebase" a term from one context to another. It corresponds directly to the renaming result from the previous chapter, but here the theorem that ensures renaming preserves typing also acts as code that performs renaming.

As before, we first need an extension lemma that allows us to extend the context when we encounter a binder. Given a map from variables in one context to variables in another, extension yields a map from the first context extended to the second context similarly extended. It looks exactly like the old extension lemma, but with all names and terms dropped:

```
ext : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ∋ A)
    ---------------------------------
  → (∀ {A B} → Γ , B ∋ A → Δ , B ∋ A)
ext ρ Z     = Z
ext ρ (S x) = S (ρ x)
```

Let `ρ` be the name of the map that takes variables in `Γ` to variables in `Δ`. Consider the de Bruijn index of the variable in `Γ , B`:

- If it is `Z`, which has type `B` in `Γ , B`, then we return `Z`, which also has type `B` in `Δ , B`.
- If it is `S x`, for some variable `x` in `Γ`, then `ρ x` is a variable in `Δ`, and hence `S (ρ x)` is a variable in `Δ , B`.

With extension under our belts, it is straightforward to define renaming. If variables in one context map to variables in another, then terms in the first context map to terms in the second:

```
rename : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ∋ A)
    -----------------------
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
rename ρ (` x)       = ` (ρ x)
rename ρ (ƛ N)       = ƛ (rename (ext ρ) N)
rename ρ (L · M)     = (rename ρ L) · (rename ρ M)
rename ρ (`zero)     = `zero
rename ρ (`suc M)    = `suc (rename ρ M)
```

```
rename ρ (case L M N) = case (rename ρ L) (rename ρ M) (rename (ext ρ) N)
rename ρ (μ N)        = μ (rename (ext ρ) N)
```

Let `ρ` be the name of the map that takes variables in `Γ` to variables in `Δ`. Let's unpack the first three cases:

- If the term is a variable, simply apply `ρ`.
- If the term is an abstraction, use the previous result to extend the map `ρ` suitably and recursively rename the body of the abstraction.
- If the term is an application, recursively rename both the function and the argument.

The remaining cases are similar, recursing on each subterm, and extending the map whenever the construct introduces a bound variable.

Whereas before renaming was a result that carried evidence that a term is well typed in one context to evidence that it is well typed in another context, now it actually transforms the term, suitably altering the bound variables. Type checking the code in Agda ensures that it is only passed and returns terms that are well typed by the rules of simply-typed lambda calculus.

Here is an example of renaming a term with one free and one bound variable:

```
M₀ : ∅ , `ℕ ⇒ `ℕ ⊢ `ℕ ⇒ `ℕ
M₀ = ƛ (# 1 · (# 1 · # 0))

M₁ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ ⇒ `ℕ
M₁ = ƛ (# 2 · (# 2 · # 0))

_ : rename S_ M₀ ≡ M₁
_ = refl
```

In general, `rename S_` will increment the de Bruijn index for each free variable by one, while leaving the index for each bound variable unchanged. The code achieves this naturally: the map originally increments each variable by one, and is extended for each bound variable by a map that leaves it unchanged.

We will see below that renaming by `S_` plays a key role in substitution. For traditional uses of de Bruijn indices without intrinsic typing, this is a little tricky. The code keeps count of a number where all greater indexes are free and all smaller indexes bound, and increment only indexes greater than the number. It's easy to have off-by-one errors. But it's hard to imagine an off-by-one error that preserves typing, and hence the Agda code for intrinsically-typed de Bruijn terms is intrinsically reliable.

Simultaneous Substitution

Because de Bruijn indices free us of concerns with renaming, it becomes easy to provide a definition of substitution that is more general than the one considered previously. Instead of substituting a closed term for a single variable, it provides a map that takes each free variable of the original term to another term. Further, the substituted terms are over an arbitrary context, and need not be closed.

The structure of the definition and the proof is remarkably close to that for renaming. Again, we first need an extension lemma that allows us to extend the context when we encounter a binder. Whereas renaming concerned a map from variables in one context to variables in another, substitution takes a map from variables in one context to *terms* in another. Given a map from

variables in one context to terms over another, extension yields a map from the first context extended to the second context similarly extended:

```
exts : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ⊢ A)
    ---------------------------------
  → (∀ {A B} → Γ , B ∋ A → Δ , B ⊢ A)
exts σ Z     = ` Z
exts σ (S x) = rename S_ (σ x)
```

Let `σ` be the name of the map that takes variables in `Γ` to terms over `Δ`. Consider the de Bruijn index of the variable in `Γ , B`:

- If it is `Z`, which has type `B` in `Γ , B`, then we return the term `` ` Z ``, which also has type `B` in `Δ , B`.
- If it is `S x`, for some variable `x` in `Γ`, then `σ x` is a term in `Δ`, and hence `rename S_ (σ x)` is a term in `Δ , B`.

This is why we had to define renaming first, since we require it to convert a term over context `Δ` to a term over the extended context `Δ , B`.

With extension under our belts, it is straightforward to define substitution. If variables in one context map to terms over another, then terms in the first context map to terms in the second:

```
subst : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ⊢ A)
    ----------------------
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
subst σ (` k)       = σ k
subst σ (ƛ N)       = ƛ (subst (exts σ) N)
subst σ (L · M)     = (subst σ L) · (subst σ M)
subst σ (`zero)     = `zero
subst σ (`suc M)    = `suc (subst σ M)
subst σ (case L M N) = case (subst σ L) (subst σ M) (subst (exts σ) N)
subst σ (μ N)       = μ (subst (exts σ) N)
```

Let `σ` be the name of the map that takes variables in `Γ` to terms over `Δ`. Let's unpack the first three cases:

- If the term is a variable, simply apply `σ`.
- If the term is an abstraction, use the previous result to extend the map `σ` suitably and recursively substitute over the body of the abstraction.
- If the term is an application, recursively substitute over both the function and the argument.

The remaining cases are similar, recursing on each subterm, and extending the map whenever the construct introduces a bound variable.

Single substitution

From the general case of substitution for multiple free variables it is easy to define the special case of substitution for one free variable:

```
_[_] : ∀ {Γ A B}
  → Γ , B ⊢ A
  → Γ ⊢ B
    ---------
  → Γ ⊢ A
_[_] {Γ} {A} {B} N M = subst {Γ , B} {Γ} σ {A} N
  where
  σ : ∀ {A} → Γ , B ∋ A → Γ ⊢ A
  σ Z     = M
  σ (S x) = ` x
```

In a term of type `A` over context `Γ , B`, we replace the variable of type `B` by a term of type `B` over context `Γ`. To do so, we use a map from the context `Γ , B` to the context `Γ`, that maps the last variable in the context to the term of type `B` and every other free variable to itself.

Consider the previous example:

- `(ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) ["s" := sucᶜ]` yields `ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")`

Here is the example formalised:

```
M₂ : ∅ , `ℕ ⇒ `ℕ ⊢ `ℕ ⇒ `ℕ
M₂ = ƛ # 1 · (# 1 · # 0)

M₃ : ∅ ⊢ `ℕ ⇒ `ℕ
M₃ = ƛ `suc # 0

M₄ : ∅ ⊢ `ℕ ⇒ `ℕ
M₄ = ƛ (ƛ `suc # 0) · ((ƛ `suc # 0) · # 0)

_ : M₂ [ M₃ ] ≡ M₄
_ = refl
```

Previously, we presented an example of substitution that we did not implement, since it needed to rename the bound variable to avoid capture:

- `(ƛ "x" ⇒ ` "x" · ` "y") [ "y" := ` "x" · `zero ]` should yield `ƛ "z" ⇒ ` "z" · (` "x" · `zero)`

Say the bound `"x"` has type `` `ℕ ⇒ `ℕ ``, the substituted `"y"` has type `` `ℕ ``, and the free `"x"` also has type `` `ℕ ⇒ `ℕ ``. Here is the example formalised:

```
M₅ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ (`ℕ ⇒ `ℕ) ⇒ `ℕ
M₅ = ƛ # 0 · # 1

M₆ : ∅ , `ℕ ⇒ `ℕ ⊢ `ℕ
M₆ = # 0 · `zero

M₇ : ∅ , `ℕ ⇒ `ℕ ⊢ (`ℕ ⇒ `ℕ) ⇒ `ℕ
M₇ = ƛ (# 0 · (# 1 · `zero))

_ : M₅ [ M₆ ] ≡ M₇
_ = refl
```

The logician Haskell Curry observed that getting the definition of substitution right can be a tricky business. It can be even trickier when using de Bruijn indices, which can often be hard to decipher.

Under the current approach, any definition of substitution must, of necessity, preserve types. While this makes the definition more involved, it means that once it is done the hardest work is out of the way. And combining definition with proof makes it harder for errors to sneak in.

Values

The definition of value is much as before, save that the added types incorporate the same information found in the Canonical Forms lemma:

```
data Value : ∀ {Γ A} → Γ ⊢ A → Set where

  V-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B}
    ---------------------------
    → Value (ƛ N)

  V-zero : ∀ {Γ}
    -----------------
    → Value (`zero {Γ})

  V-suc : ∀ {Γ} {V : Γ ⊢ `ℕ}
    → Value V
      --------------
    → Value (`suc V)
```

Here `zero` requires an implicit parameter to aid inference, much in the same way that `[]` did in Lists.

Reduction

The reduction rules are the same as those given earlier, save that for each term we must specify its types. As before, we have compatibility rules that reduce a part of a term, labelled with `ξ`, and rules that simplify a constructor combined with a destructor, labelled with `β`:

```
infix 2 _⟶_

data _⟶_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  ξ-·₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A ⇒ B} {M : Γ ⊢ A}
    → L ⟶ L′
      ---------------
    → L · M ⟶ L′ · M

  ξ-·₂ : ∀ {Γ A B} {V : Γ ⊢ A ⇒ B} {M M′ : Γ ⊢ A}
    → Value V
    → M ⟶ M′
      ---------------
    → V · M ⟶ V · M′

  β-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B} {W : Γ ⊢ A}
    → Value W
      --------------------
    → (ƛ N) · W ⟶ N [ W ]

  ξ-suc : ∀ {Γ} {M M′ : Γ ⊢ `ℕ}
    → M ⟶ M′
```

```
    -----------------
  → `suc M ⟶ `suc M′

ξ-case : ∀ {Γ A} {L L′ : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
  → L ⟶ L′
    ------------------------
  → case L M N ⟶ case L′ M N

β-zero : ∀ {Γ A} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
    -------------------
  → case `zero M N ⟶ M

β-suc : ∀ {Γ A} {V : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
  → Value V
    ----------------------------
  → case (`suc V) M N ⟶ N [ V ]

β-μ : ∀ {Γ A} {N : Γ , A ⊢ A}
    ----------------
  → μ N ⟶ N [ μ N ]
```

The definition states that `M ⟶ N` can only hold of terms `M` and `N` which *both* have type `Γ ⊢ A` for some context `Γ` and type `A`. In other words, it is *built-in* to our definition that reduction preserves types. There is no separate Preservation theorem to prove. The Agda type-checker validates that each term preserves types. In the case of `β` rules, preservation depends on the fact that substitution preserves types, which is built-in to our definition of substitution.

Reflexive and transitive closure

The reflexive and transitive closure is exactly as before. We simply cut-and-paste the previous definition:

```
infix 2 _⟶↠_
infix 1 begin_
infixr 2 _⟶⟨_⟩_
infix 3 _∎

data _⟶↠_ {Γ A} : (Γ ⊢ A) → (Γ ⊢ A) → Set where

  _∎ : (M : Γ ⊢ A)
      ------
    → M ⟶↠ M

  _⟶⟨_⟩_ : (L : Γ ⊢ A) {M N : Γ ⊢ A}
    → L ⟶ M
    → M ⟶↠ N
      ------
    → L ⟶↠ N

begin_ : ∀ {Γ A} {M N : Γ ⊢ A}
  → M ⟶↠ N
    ------
  → M ⟶↠ N
begin M⟶↠N = M⟶↠N
```

Examples

We reiterate each of our previous examples. First, the Church numeral two applied to the successor function and zero yields the natural number two:

```
_ : twoᶜ · sucᶜ · `zero {∅} —↠ `suc `suc `zero
_ =
  begin
    twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ (sucᶜ · (sucᶜ · # 0))) · `zero
  —→⟨ β-ƛ V-zero ⟩
    sucᶜ · (sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    sucᶜ · `suc `zero
  —→⟨ β-ƛ (V-suc V-zero) ⟩
    `suc (`suc `zero)
  ∎
```

As before, we need to supply an explicit context to `` `zero ``.

Next, a sample reduction demonstrating that two plus two is four:

```
_ : plus {∅} · two · two —↠ `suc `suc `suc `suc `zero
_ =
    plus · two · two
  —→⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
    (ƛ ƛ case (` S Z) (` Z) (`suc (plus · ` Z · ` S Z))) · two · two
  —→⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ case two (` Z) (`suc (plus · ` Z · ` S Z))) · two
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    case two two (`suc (plus · ` Z · two))
  —→⟨ β-suc (V-suc V-zero) ⟩
    `suc (plus · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
    `suc ((ƛ ƛ case (` S Z) (` Z) (`suc (plus · ` Z · ` S Z)))
      · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
    `suc ((ƛ case (`suc `zero) (` Z) (`suc (plus · ` Z · ` S Z))) · two)
  —→⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
    `suc (case (`suc `zero) (two) (`suc (plus · ` Z · two)))
  —→⟨ ξ-suc (β-suc V-zero) ⟩
    `suc (`suc (plus · `zero · two))
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
    `suc (`suc ((ƛ ƛ case (` S Z) (` Z) (`suc (plus · ` Z · ` S Z)))
      · `zero · two))
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
    `suc (`suc ((ƛ case `zero (` Z) (`suc (plus · ` Z · ` S Z))) · two))
  —→⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
    `suc (`suc (case `zero (two) (`suc (plus · ` Z · two))))
  —→⟨ ξ-suc (ξ-suc β-zero) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

And finally, a similar sample reduction for Church numerals:

```
_ : plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero —↠ `suc `suc `suc `suc `zero {∅}
_ =
  begin
    plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
    (ƛ ƛ ƛ twoᶜ · ` S Z · (` S S Z · ` S Z · ` Z)) · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ ƛ twoᶜ · ` S Z · (twoᶜ · ` S Z · ` Z)) · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ twoᶜ · sucᶜ · (twoᶜ · sucᶜ · ` Z)) · `zero
  —→⟨ β-ƛ V-zero ⟩
    twoᶜ · sucᶜ · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · ((ƛ sucᶜ · (sucᶜ · ` Z)) · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · (sucᶜ · (sucᶜ · `zero))
  —→⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · (sucᶜ · `suc `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · `suc (`suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    sucᶜ · (sucᶜ · `suc (`suc `zero))
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    sucᶜ · `suc (`suc (`suc `zero))
  —→⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

Values do not reduce

We have now completed all the definitions, which of necessity subsumed some of the propositions from the earlier development: Canonical Forms, Substitution preserves types, and Preservation. We now turn to proving the remaining results from the previous development.

Exercise `V¬—→` (practice)

Following the previous development, show values do not reduce, and its corollary, terms that reduce are not values.

```
-- Your code goes here
```

Progress

As before, every term that is well typed and closed is either a value or takes a reduction step. The formulation of progress is just as before, but annotated with types:

```
data Progress {A} (M : ∅ ⊢ A) : Set where
```

```
  step : ∀ {N : ∅ ⊢ A}
    → M ⟶ N
      ----------
    → Progress M

  done :
      Value M
      ----------
    → Progress M
```

The statement and proof of progress is much as before, appropriately annotated. We no longer need to explicitly refer to the Canonical Forms lemma, since it is built-in to the definition of value:

```
progress : ∀ {A} → (M : ∅ ⊢ A) → Progress M
progress (` ())
progress (ƛ N)        = done V-ƛ
progress (L · M) with progress L
... | step L⟶L′       = step (ξ-·₁ L⟶L′)
... | done V-ƛ with progress M
... | step M⟶M′       = step (ξ-·₂ V-ƛ M⟶M′)
... | done VM         = step (β-ƛ VM)
progress (`zero)      = done V-zero
progress (`suc M) with progress M
... | step M⟶M′       = step (ξ-suc M⟶M′)
... | done VM         = done (V-suc VM)
progress (case L M N) with progress L
... | step L⟶L′       = step (ξ-case L⟶L′)
... | done V-zero     = step (β-zero)
... | done (V-suc VL) = step (β-suc VL)
progress (μ N)        = step (β-μ)
```

Evaluation

Before, we combined progress and preservation to evaluate a term. We can do much the same here, but we no longer need to explicitly refer to preservation, since it is built-in to the definition of reduction.

As previously, gas is specified by a natural number:

```
record Gas : Set where
  constructor gas
  field
    amount : ℕ
```

When our evaluator returns a term `N`, it will either give evidence that `N` is a value or indicate that it ran out of gas:

```
data Finished {Γ A} (N : Γ ⊢ A) : Set where

  done :
    Value N
    ----------
    → Finished N

  out-of-gas :
    ----------
```

```
    Finished N
```

Given a term `L` of type `A`, the evaluator will, for some `N`, return a reduction sequence from `L` to `N` and an indication of whether reduction finished:

```
data Steps {A} : ∅ ⊢ A → Set where

  steps : {L N : ∅ ⊢ A}
    → L —↠ N
    → Finished N
      ----------
    → Steps L
```

The evaluator takes gas and a term and returns the corresponding steps:

```
eval : ∀ {A}
  → Gas
  → (L : ∅ ⊢ A)
    -----------
  → Steps L
eval (gas zero) L    = steps (L ∎) out-of-gas
eval (gas (suc m)) L with progress L
... | done VL        = steps (L ∎) (done VL)
... | step {M} L—→M with eval (gas m) M
... | steps M—↠N fin = steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

The definition is a little simpler than previously, as we no longer need to invoke preservation.

Examples

We reiterate each of our previous examples. We re-define the term `sucμ` that loops forever:

```
sucμ : ∅ ⊢ `ℕ
sucμ = μ (`suc (# 0))
```

To compute the first three steps of the infinite reduction sequence, we evaluate with three steps worth of gas:

```
_ : eval (gas 3) sucμ ≡
  steps
    (μ `suc ` Z
    —→⟨ β-μ ⟩
      `suc (μ `suc ` Z)
    —→⟨ ξ-suc β-μ ⟩
      `suc (`suc (μ `suc ` Z))
    —→⟨ ξ-suc (ξ-suc β-μ) ⟩
      `suc (`suc (`suc (μ `suc ` Z)))
    ∎)
    out-of-gas
_ = refl
```

The Church numeral two applied to successor and zero:

```
_ : eval (gas 100) (twoᶜ · sucᶜ · `zero) ≡
  steps
    ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · `zero
    ⟶⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
      (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) · `zero
    ⟶⟨ β-ƛ V-zero ⟩
      (ƛ `suc ` Z) · ((ƛ `suc ` Z) · `zero)
    ⟶⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
      (ƛ `suc ` Z) · `suc `zero
    ⟶⟨ β-ƛ (V-suc V-zero) ⟩
      `suc (`suc `zero)
    ∎)
    (done (V-suc (V-suc V-zero)))
_ = refl
```

Two plus two is four:

```
_ : eval (gas 100) (plus · two · two) ≡
  steps
    ((μ
       (ƛ
         (ƛ
           case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
      · `suc (`suc `zero)
      · `suc (`suc `zero)
    ⟶⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
      (ƛ
        (ƛ
          case (` (S Z)) (` Z)
          (`suc
            ((μ
               (ƛ
                 (ƛ
                   case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
              · ` Z
              · ` (S Z)))))
      · `suc (`suc `zero)
      · `suc (`suc `zero)
    ⟶⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
      (ƛ
       case (`suc (`suc `zero)) (` Z)
       (`suc
         ((μ
            (ƛ
              (ƛ
                case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
           · ` Z
           · ` (S Z))))
      · `suc (`suc `zero)
    ⟶⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
      case (`suc (`suc `zero)) (`suc (`suc `zero))
      (`suc
        ((μ
           (ƛ
             (ƛ
               case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
          · ` Z
          · `suc (`suc `zero)))
    ⟶⟨ β-suc (V-suc V-zero) ⟩
```

```
    `suc
    ((μ
       (ƛ
         (ƛ
           case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
      · `suc `zero
      · `suc (`suc `zero))
  ⟶⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
    `suc
    ((ƛ
       (ƛ
         case (` (S Z)) (` Z)
         (`suc
           ((μ
              (ƛ
                (ƛ
                  case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
             · ` Z
             · ` (S Z)))))
      · `suc `zero
      · `suc (`suc `zero))
  ⟶⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
    `suc
    ((ƛ
       case (`suc `zero) (` Z)
       (`suc
         ((μ
            (ƛ
              (ƛ
                case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
           · ` Z
           · ` (S Z))))
      · `suc (`suc `zero))
  ⟶⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
    `suc
    case (`suc `zero) (`suc (`suc `zero))
    (`suc
      ((μ
         (ƛ
           (ƛ
             case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
        · ` Z
        · `suc (`suc `zero)))
  ⟶⟨ ξ-suc (β-suc V-zero) ⟩
    `suc
    (`suc
      ((μ
         (ƛ
           (ƛ
             case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
        · `zero
        · `suc (`suc `zero)))
  ⟶⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
    `suc
    (`suc
      ((ƛ
         (ƛ
           case (` (S Z)) (` Z)
           (`suc
```

```
            ((μ
               (ƛ
                 (ƛ
                   case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
              · ` Z
              · ` (S Z)))))
        · `zero
        · `suc (`suc `zero)))
  ⟶⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
    `suc
    (`suc
     ((ƛ
        case `zero (` Z)
        (`suc
          ((μ
             (ƛ
               (ƛ
                 case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
            · ` Z
            · ` (S Z))))
       · `suc (`suc `zero)))
  ⟶⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
    `suc
    (`suc
     case `zero (`suc (`suc `zero))
     (`suc
       ((μ
          (ƛ
            (ƛ
              case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
         · ` Z
         · `suc (`suc `zero))))
  ⟶⟨ ξ-suc (ξ-suc β-zero) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎)
  (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

And the corresponding term for Church numerals:

```
_ : eval (gas 100) (plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero) ≡
  steps
    ((ƛ
       (ƛ
         (ƛ (ƛ ` (S (S (S Z))) · ` (S Z) · (` (S (S Z)) · ` (S Z) · ` Z)))))
      · (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z)))
      · (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z)))
      · (ƛ `suc ` Z)
      · `zero
    ⟶⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
      (ƛ
        (ƛ
          (ƛ
            (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · ` (S Z) ·
            (` (S (S Z)) · ` (S Z) · ` Z))))
      · (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z)))
      · (ƛ `suc ` Z)
      · `zero
    ⟶⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
```

```
    (ƛ
      (ƛ
        (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · ` (S Z) ·
        ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · ` (S Z) · ` Z)))
    · (ƛ `suc ` Z)
    · `zero
  ⟶⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ
      (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) ·
      ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · ` Z))
    · `zero
  ⟶⟨ β-ƛ V-zero ⟩
    (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) ·
    ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · `zero)
  ⟶⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · `zero)
  ⟶⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) · `zero)
  ⟶⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ `suc ` Z) · ((ƛ `suc ` Z) · `zero))
  ⟶⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ `suc ` Z) · `suc `zero)
  ⟶⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) · `suc (`suc `zero)
  ⟶⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    (ƛ `suc ` Z) · ((ƛ `suc ` Z) · `suc (`suc `zero))
  ⟶⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ `suc ` Z) · `suc (`suc (`suc `zero))
  ⟶⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎)
  (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

We omit the proof that reduction is deterministic, since it is tedious and almost identical to the previous proof.

Exercise `mul-example` (recommended)

Using the evaluator, confirm that two times two is four.

```
-- Your code goes here
```

Intrinsic typing is golden

Counting the lines of code is instructive. While this chapter covers the same formal development as the previous two chapters, it has much less code. Omitting all the examples, and all proofs that appear in Properties but not DeBruijn (such as the proof that reduction is deterministic), the number of lines of code is as follows:

```
Lambda                  216
Properties              235
DeBruijn                276
```

The relation between the two approaches approximates the golden ratio: extrinsically-typed terms require about 1.6 times as much code as intrinsically-typed.

Unicode

This chapter uses the following unicode:

```
σ  U+03C3  GREEK SMALL LETTER SIGMA (\Gs or \sigma)
₀  U+2080  SUBSCRIPT ZERO (\_0)
₃  U+20B3  SUBSCRIPT THREE (\_3)
₄  U+2084  SUBSCRIPT FOUR (\_4)
₅  U+2085  SUBSCRIPT FIVE (\_5)
₆  U+2086  SUBSCRIPT SIX (\_6)
₇  U+2087  SUBSCRIPT SEVEN (\_7)
≠  U+2260  NOT EQUAL TO (\=n)
```

```
mul : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
mul = μ ✗ ✗ (case (# 1) `zero (plus · # 1 · (# 3 · # 0 · # 1)))
```

_ : eval (gas 100) (mul · two · two) ≡ [code generated by ctrl+c ctrl+n]

_ = refl

Chapter 14

More: Additional constructs of simply-typed lambda calculus

```
module plfa.part2.More where
```

So far, we have focussed on a relatively minimal language, based on Plotkin's PCF, which supports functions, naturals, and fixpoints. In this chapter we extend our calculus to support the following:

- primitive numbers
- *let* bindings
- products
- an alternative formulation of products
- sums
- unit type
- an alternative formulation of unit type
- empty type
- lists

All of the data types should be familiar from Part I of this textbook. For *let* and the alternative formulations we show how they translate to other constructs in the calculus. Most of the description will be informal. We show how to formalise the first four constructs and leave the rest as an exercise for the reader.

Our informal descriptions will be in the style of Chapter Lambda, using extrinsically-typed terms, while our formalisation will be in the style of Chapter DeBruijn, using intrinsically-typed terms.

By now, explaining with symbols should be more concise, more precise, and easier to follow than explaining in prose. For each construct, we give syntax, typing, reductions, and an example. We also give translations where relevant; formally establishing the correctness of translations will be the subject of the next chapter.

Primitive numbers

We define a `Nat` type equivalent to the built-in natural number type with multiplication as a primitive operation on numbers:

Syntax

```
A, B, C ::= ...                    Types
  Nat                                primitive natural numbers

L, M, N ::= ...                    Terms
  con c                              constant
  L `* M                             multiplication

V, W ::= ...                       Values
  con c                              constant
```

Typing

The hypothesis of the `con` rule is unusual, in that it refers to a typing judgment of Agda rather than a typing judgment of the defined calculus:

```
c : ℕ
--------------- con
Γ ⊢ con c : Nat

Γ ⊢ L : Nat
Γ ⊢ M : Nat
--------------- _`*_
Γ ⊢ L `* M : Nat
```

Reduction

A rule that defines a primitive directly, such as the last rule below, is called a δ rule. Here the δ rule defines multiplication of primitive numbers in terms of multiplication of naturals as given by the Agda standard prelude:

```
L ⟶ L′
---------------- ξ-*₁
L `* M ⟶ L′ `* M

M ⟶ M′
---------------- ξ-*₂
V `* M ⟶ V `* M′

---------------------------- δ-*
con c `* con d ⟶ con (c * d)
```

Example

Here is a function to cube a primitive number:

```
cube : ∅ ⊢ Nat ⇒ Nat
cube = ƛ x ⇒ x `* x `* x
```

Let bindings

Let bindings affect only the syntax of terms; they introduce no new types or values:

Syntax

```
L, M, N ::= ...                    Terms
  `let x `= M `in N                  let
```

Typing

```
Γ ⊢ M ⦂ A
Γ , x ⦂ A ⊢ N ⦂ B
------------------------ `let
Γ ⊢ `let x `= M `in N ⦂ B
```

Reduction

```
M ⟶ M′
-------------------------------------- ξ-let
`let x `= M `in N ⟶ `let x `= M′ `in N

--------------------------------- β-let
`let x `= V `in N ⟶ N [ x := V ]
```

Example

Here is a function to raise a primitive number to the tenth power:

```
exp10 : ∅ ⊢ Nat ⇒ Nat
exp10 = ƛ x ⇒ `let x2  `= x  `* x  `in
              `let x4  `= x2 `* x2 `in
              `let x5  `= x4 `* x  `in
              x5 `* x5
```

Translation

We can translate each *let* term into an application of an abstraction:

```
(`let x `= M `in N) †  =  (ƛ x ⇒ (N †)) · (M †)
```

Here `M †` is the translation of term `M` from a calculus with the construct to a calculus without the construct.

Products

Syntax

```
A, B, C ::= ...                    Types
  A `× B                             product type

L, M, N ::= ...                    Terms
  `⟨ M , N ⟩                         pair
  `proj₁ L                           project first component
  `proj₂ L                           project second component

V, W ::= ...                       Values
  `⟨ V , W ⟩                         pair
```

Typing

```
Γ ⊢ M ⦂ A
Γ ⊢ N ⦂ B
----------------------- `⟨_,_⟩ or `×-I
Γ ⊢ `⟨ M , N ⟩ ⦂ A `× B

Γ ⊢ L ⦂ A `× B
---------------- `proj₁ or `×-E₁
Γ ⊢ `proj₁ L ⦂ A

Γ ⊢ L ⦂ A `× B
---------------- `proj₂ or `×-E₂
Γ ⊢ `proj₂ L ⦂ B
```

Reduction

```
M ⟶ M′
------------------------- ξ-⟨,⟩₁
`⟨ M , N ⟩ ⟶ `⟨ M′ , N ⟩

N ⟶ N′
------------------------- ξ-⟨,⟩₂
`⟨ V , N ⟩ ⟶ `⟨ V , N′ ⟩

L ⟶ L′
--------------------- ξ-proj₁
`proj₁ L ⟶ `proj₁ L′

L ⟶ L′
--------------------- ξ-proj₂
`proj₂ L ⟶ `proj₂ L′

---------------------- β-proj₁
`proj₁ `⟨ V , W ⟩ ⟶ V
```

```
---------------------- β-proj₂
`proj₂ `⟨ V , W ⟩ ⟶ W
```

Example

Here is a function to swap the components of a pair:

```
swap× : ∅ ⊢ A `× B ⇒ B `× A
swap× = ƛ z ⇒ `⟨ `proj₂ z , `proj₁ z ⟩
```

Alternative formulation of products

There is an alternative formulation of products, where in place of two ways to eliminate the type we have a case term that binds two variables. We repeat the syntax in full, but only give the new type and reduction rules:

Syntax

```
A, B, C ::= ...                    Types
  A `× B                             product type

L, M, N ::= ...                    Terms
  `⟨ M , N ⟩                         pair
  case× L [⟨ x , y ⟩⇒ M ]             case

V, W ::=                           Values
  `⟨ V , W ⟩                         pair
```

Typing

```
Γ ⊢ L ⦂ A `× B
Γ , x ⦂ A , y ⦂ B ⊢ N ⦂ C
------------------------------ case× or ×-E
Γ ⊢ case× L [⟨ x , y ⟩⇒ N ] ⦂ C
```

Reduction

```
L ⟶ L′
------------------------------------------------ ξ-case×
case× L [⟨ x , y ⟩⇒ N ] ⟶ case× L′ [⟨ x , y ⟩⇒ N ]

---------------------------------------------------------- β-case×
case× `⟨ V , W ⟩ [⟨ x , y ⟩⇒ N ] ⟶ N [ x := V ][ y := W ]
```

Example

Here is a function to swap the components of a pair rewritten in the new notation:

```
swap×-case : ∅ ⊢ A `× B ⇒ B `× A
swap×-case = ƛ z ⇒ case× z
                    [⟨ x , y ⟩⇒ `⟨ y , x ⟩ ]
```

Translation

We can translate the alternative formulation into the one with projections:

```
  (case× L [⟨ x , y ⟩⇒ N ]) †
=
  `let z `= (L †) `in
  `let x `= `proj₁ z `in
  `let y `= `proj₂ z `in
  (N †)
```

Here `z` is a variable that does not appear free in `N`. We refer to such a variable as *fresh*.

One might think that we could instead use a more compact translation:

```
-- WRONG
  (case× L [⟨ x , y ⟩⇒ N ]) †
=
  (N †) [ x := `proj₁ (L †) ] [ y := `proj₂ (L †) ]
```

But this behaves differently. The first term always reduces `L` before `N`, and it computes `proj₁`` and `proj₂` exactly once. The second term does not reduce `L` to a value before reducing `N` and `proj₂` many times or not at all.

We can also translate back the other way:

```
(`proj₁ L) ‡  =  case× (L ‡) [⟨ x , y ⟩⇒ x ]
(`proj₂ L) ‡  =  case× (L ‡) [⟨ x , y ⟩⇒ y ]
```

Sums

Syntax

```
A, B, C ::= ...                      Types
  A `⊎ B                               sum type

L, M, N ::= ...                      Terms
  `inj₁ M                              inject first component
  `inj₂ N                              inject second component
  case⊎ L [inj₁ x ⇒ M |inj₂ y ⇒ N ]     case

V, W ::= ...                         Values
```

```
  `inj₁ V                          inject first component
  `inj₂ W                          inject second component
```

Typing

```
Γ ⊢ M ⦂ A
-------------------- `inj₁ or ⊎-I₁
Γ ⊢ `inj₁ M ⦂ A `⊎ B

Γ ⊢ N ⦂ B
-------------------- `inj₂ or ⊎-I₂
Γ ⊢ `inj₂ N ⦂ A `⊎ B

Γ ⊢ L ⦂ A `⊎ B
Γ , x ⦂ A ⊢ M ⦂ C
Γ , y ⦂ B ⊢ N ⦂ C
---------------------------------------- case⊎ or ⊎-E
Γ ⊢ case⊎ L [inj₁ x ⇒ M |inj₂ y ⇒ N ] ⦂ C
```

Reduction

```
M ⟶ M′
------------------- ξ-inj₁
`inj₁ M ⟶ `inj₁ M′

N ⟶ N′
------------------- ξ-inj₂
`inj₂ N ⟶ `inj₂ N′

L ⟶ L′
------------------------------------------------------------------------ ξ-case⊎
case⊎ L [inj₁ x ⇒ M |inj₂ y ⇒ N ] ⟶ case⊎ L′ [inj₁ x ⇒ M |inj₂ y ⇒ N ]

---------------------------------------------------------- β-inj₁
case⊎ (`inj₁ V) [inj₁ x ⇒ M |inj₂ y ⇒ N ] ⟶ M [ x := V ]

---------------------------------------------------------- β-inj₂
case⊎ (`inj₂ W) [inj₁ x ⇒ M |inj₂ y ⇒ N ] ⟶ N [ y := W ]
```

Example

Here is a function to swap the components of a sum:

```
swap⊎ : ∅ ⊢ A `⊎ B ⇒ B `⊎ A
swap⊎ = ƛ z ⇒ case⊎ z
                [inj₁ x ⇒ `inj₂ x
                |inj₂ y ⇒ `inj₁ y ]
```

Unit type

For the unit type, there is a way to introduce values of the type but no way to eliminate values of the type. There are no reduction rules.

Syntax

```
A, B, C ::= ...                    Types
  `⊤                                 unit type

L, M, N ::= ...                    Terms
  `tt                                unit value

V, W ::= ...                       Values
  `tt                                unit value
```

Typing

```
------------ `tt or ⊤-I
Γ ⊢ `tt ⦂ `⊤
```

Reduction

(none)

Example

Here is the isomorphism between `A` and `A `× `⊤`:

```
to×⊤ : ∅ ⊢ A ⇒ A `× `⊤
to×⊤ = ƛ x ⇒ `⟨ x , `tt ⟩

from×⊤ : ∅ ⊢ A `× `⊤ ⇒ A
from×⊤ = ƛ z ⇒ `proj₁ z
```

Alternative formulation of unit type

There is an alternative formulation of the unit type, where in place of no way to eliminate the type we have a case term that binds zero variables. We repeat the syntax in full, but only give the new type and reduction rules:

Syntax

```
A, B, C ::= ...                     Types
  `⊤                                  unit type

L, M, N ::= ...                     Terms
  `tt                                 unit value
  `case⊤ L [tt⇒ N ]                   case

V, W ::= ...                        Values
  `tt                                 unit value
```

Typing

```
Γ ⊢ L ⇇ `⊤
Γ ⊢ M ⇇ A
----------------------- case⊤ or ⊤-E
Γ ⊢ case⊤ L [tt⇒ M ] ⇇ A
```

Reduction

```
L ⟶ L′
------------------------------------ ξ-case⊤
case⊤ L [tt⇒ M ] ⟶ case⊤ L′ [tt⇒ M ]

---------------------- β-case⊤
case⊤ `tt [tt⇒ M ] ⟶ M
```

Example

Here is half the isomorphism between `A` and `A `× `⊤` rewritten in the new notation:

```
from×⊤-case : ∅ ⊢ A `× `⊤ ⇒ A
from×⊤-case = ƛ z ⇒ case× z
                     [⟨ x , y ⟩⇒ case⊤ y
                                   [tt⇒ x ] ]
```

Translation

We can translate the alternative formulation into one without case:

```
(case⊤ L [tt⇒ M ]) †  =  `let z `= (L †) `in (M †)
```

Here `z` is a variable that does not appear free in `M`.

Empty type

For the empty type, there is a way to eliminate values of the type but no way to introduce values of the type. There are no values of the type and no β rule, but there is a ξ rule. The `case⊥` construct plays a role similar to `⊥-elim` in Agda:

Syntax

```
A, B, C ::= ...                    Types
  `⊥                                 empty type

L, M, N ::= ...                    Terms
  case⊥ L []                         case
```

Typing

```
Γ ⊢ L ⇇ `⊥
------------------ case⊥ or ⊥-E
Γ ⊢ case⊥ L [] ⇇ A
```

Reduction

```
L ⟶ L′
------------------------ ξ-case⊥
case⊥ L [] ⟶ case⊥ L′ []
```

Example

Here is the isomorphism between `A` and `A `⊎ `⊥`:

```
to⊎⊥ : ∅ ⊢ A ⇒ A `⊎ `⊥
to⊎⊥ = ƛ x ⇒ `inj₁ x

from⊎⊥ : ∅ ⊢ A `⊎ `⊥ ⇒ A
from⊎⊥ = ƛ z ⇒ case⊎ z
                 [inj₁ x ⇒ x
                 |inj₂ y ⇒ case⊥ y
                             [] ]
```

Lists

Syntax

```
A, B, C ::= ...                     Types
  `List A                             list type

L, M, N ::= ...                     Terms
  `[]                                 nil
  M `:: N                             cons
  caseL L [[]⇒ M | x :: y ⇒ N ]       case

V, W ::= ...                        Values
  `[]                                 nil
  V `:: W                             cons
```

Typing

```
----------------- `[] or List-I₁
Γ ⊢ `[] ⦂ `List A

Γ ⊢ M ⦂ A
Γ ⊢ N ⦂ `List A
------------------- _`::_ or List-I₂
Γ ⊢ M `:: N ⦂ `List A

Γ ⊢ L ⦂ `List A
Γ ⊢ M ⦂ B
Γ , x ⦂ A , xs ⦂ `List A ⊢ N ⦂ B
------------------------------------ caseL or List-E
Γ ⊢ caseL L [[]⇒ M | x :: xs ⇒ N ] ⦂ B
```

Reduction

```
M ⟶ M′
---------------- ξ-::₁
M `:: N ⟶ M′ `:: N

N ⟶ N′
---------------- ξ-::₂
V `:: N ⟶ V `:: N′

L ⟶ L′
------------------------------------------------------------- ξ-caseL
caseL L [[]⇒ M | x :: xs ⇒ N ] ⟶ caseL L′ [[]⇒ M | x :: xs ⇒ N ]

---------------------------------- β-[]
caseL `[] [[]⇒ M | x :: xs ⇒ N ] ⟶ M

------------------------------------------------------------- β-::
caseL (V `:: W) [[]⇒ M | x :: xs ⇒ N ] ⟶ N [ x := V ][ xs := W ]
```

Example

Here is the map function for lists:

```
mapL : ∅ ⊢ (A ⇒ B) ⇒ `List A ⇒ `List B
mapL = μ mL ⇒ ƛ f ⇒ ƛ xs ⇒
         caseL xs
           [[]⇒ `[]
           | x :: xs ⇒ f · x `:: mL · f · xs ]
```

Formalisation

We now show how to formalise

- primitive numbers
- *let* bindings
- products
- an alternative formulation of products

and leave formalisation of the remaining constructs as an exercise.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _*_; _<_; _≤?_; z≤n; s≤s)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Decidable using (True; toWitness)
```

Syntax

```
infix 4 _⊢_
infix 4 _∋_
infixl 5 _,_

infixr 7 _⇒_
infixr 9 _`×_

infix 5 ƛ_
infix 5 μ_
infixl 7 _·_
infixl 8 _`*_
infix 8 `suc_
infix 9 `_
infix 9 S_
infix 9 #_
```

Types

```
data Type : Set where
  `ℕ    : Type
  _⇒_   : Type → Type → Type
  Nat   : Type
  _`×_  : Type → Type → Type
```

Contexts

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

Variables and the lookup judgment

```
data _∋_ : Context → Type → Set where

  Z : ∀ {Γ A}
      ---------
    → Γ , A ∋ A

  S_ : ∀ {Γ A B}
    → Γ ∋ B
      ---------
    → Γ , A ∋ B
```

Terms and the typing judgment

```
data _⊢_ : Context → Type → Set where

  -- variables

  `_ : ∀ {Γ A}
    → Γ ∋ A
      -----
    → Γ ⊢ A

  -- functions

  ƛ_ : ∀ {Γ A B}
    → Γ , A ⊢ B
      ---------
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
      ---------
    → Γ ⊢ B
```

```
-- naturals

`zero : ∀ {Γ}
    ------
  → Γ ⊢ `ℕ

`suc_ : ∀ {Γ}
  → Γ ⊢ `ℕ
    ------
  → Γ ⊢ `ℕ

case : ∀ {Γ A}
  → Γ ⊢ `ℕ
  → Γ ⊢ A
  → Γ , `ℕ ⊢ A
    -----
  → Γ ⊢ A

-- fixpoint

μ_ : ∀ {Γ A}
  → Γ , A ⊢ A
    ----------
  → Γ ⊢ A

-- primitive numbers

con : ∀ {Γ}
  → ℕ
    -------
  → Γ ⊢ Nat

_`*_ : ∀ {Γ}
  → Γ ⊢ Nat
  → Γ ⊢ Nat
    -------
  → Γ ⊢ Nat

-- let

`let : ∀ {Γ A B}
  → Γ ⊢ A
  → Γ , A ⊢ B
    ----------
  → Γ ⊢ B

-- products

`⟨_,_⟩ : ∀ {Γ A B}
  → Γ ⊢ A
  → Γ ⊢ B
    -----------
  → Γ ⊢ A `× B

`proj₁ : ∀ {Γ A B}
  → Γ ⊢ A `× B
    -----------
  → Γ ⊢ A

`proj₂ : ∀ {Γ A B}
  → Γ ⊢ A `× B
    -----------
```

```
    → Γ ⊢ B

  -- alternative formulation of products

  case× : ∀ {Γ A B C}
    → Γ ⊢ A `× B
    → Γ , A , B ⊢ C
      --------------
    → Γ ⊢ C
```

Abbreviating de Bruijn indices

```
length : Context → ℕ
length ∅       = zero
length (Γ , _) = suc (length Γ)

lookup : {Γ : Context} → {n : ℕ} → (p : n < length Γ) → Type
lookup {(_ , A)} {zero} (s≤s z≤n) = A
lookup {(Γ , _)} {(suc n)} (s≤s p) = lookup p

count : ∀ {Γ} → {n : ℕ} → (p : n < length Γ) → Γ ∋ lookup p
count {_ , _} {zero} (s≤s z≤n) =  Z
count {Γ , _} {(suc n)} (s≤s p) =  S (count p)

#_ : ∀ {Γ}
  → (n : ℕ)
  → {n∈Γ : True (suc n ≤? length Γ)}
    --------------------------------
  → Γ ⊢ lookup (toWitness n∈Γ)
#_ n {n∈Γ} = ` count (toWitness n∈Γ)
```

Renaming

```
ext : ∀ {Γ Δ}
  → (∀ {A}  → Γ ∋ A → Δ ∋ A)
    --------------------------------
  → (∀ {A B} → Γ , A ∋ B → Δ , A ∋ B)
ext ρ Z     = Z
ext ρ (S x) = S (ρ x)

rename : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ∋ A)
    ----------------------
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
rename ρ (` x)        = ` (ρ x)
rename ρ (ƛ N)        = ƛ (rename (ext ρ) N)
rename ρ (L · M)      = (rename ρ L) · (rename ρ M)
rename ρ (`zero)      = `zero
rename ρ (`suc M)     = `suc (rename ρ M)
rename ρ (case L M N) = case (rename ρ L) (rename ρ M) (rename (ext ρ) N)
rename ρ (μ N)        = μ (rename (ext ρ) N)
rename ρ (con n)      = con n
```

```
rename ρ (M `* N)     = rename ρ M `* rename ρ N
rename ρ (`let M N)   = `let (rename ρ M) (rename (ext ρ) N)
rename ρ `⟨ M , N ⟩   = `⟨ rename ρ M , rename ρ N ⟩
rename ρ (`proj₁ L)   = `proj₁ (rename ρ L)
rename ρ (`proj₂ L)   = `proj₂ (rename ρ L)
rename ρ (case× L M)  = case× (rename ρ L) (rename (ext (ext ρ)) M)
```

Simultaneous Substitution

```
exts : ∀ {Γ Δ} → (∀ {A} → Γ ∋ A → Δ ⊢ A) → (∀ {A B} → Γ , A ∋ B → Δ , A ⊢ B)
exts σ Z     = ` Z
exts σ (S x) = rename S_ (σ x)

subst : ∀ {Γ Δ} → (∀ {C} → Γ ∋ C → Δ ⊢ C) → (∀ {C} → Γ ⊢ C → Δ ⊢ C)
subst σ (` k)        = σ k
subst σ (ƛ N)        = ƛ (subst (exts σ) N)
subst σ (L · M)      = (subst σ L) · (subst σ M)
subst σ (`zero)      = `zero
subst σ (`suc M)     = `suc (subst σ M)
subst σ (case L M N) = case (subst σ L) (subst σ M) (subst (exts σ) N)
subst σ (μ N)        = μ (subst (exts σ) N)
subst σ (con n)      = con n
subst σ (M `* N)     = subst σ M `* subst σ N
subst σ (`let M N)   = `let (subst σ M) (subst (exts σ) N)
subst σ `⟨ M , N ⟩   = `⟨ subst σ M , subst σ N ⟩
subst σ (`proj₁ L)   = `proj₁ (subst σ L)
subst σ (`proj₂ L)   = `proj₂ (subst σ L)
subst σ (case× L M)  = case× (subst σ L) (subst (exts (exts σ)) M)
```

Single and double substitution

```
substZero : ∀ {Γ}{A B} → Γ ⊢ A → Γ , A ∋ B → Γ ⊢ B
substZero V Z     = V
substZero V (S x) = ` x

_[_] : ∀ {Γ A B}
  → Γ , A ⊢ B
  → Γ ⊢ A
    ---------
  → Γ ⊢ B
_[_] {Γ} {A} N V = subst {Γ , A} {Γ} (substZero V) N

_[_][_] : ∀ {Γ A B C}
  → Γ , A , B ⊢ C
  → Γ ⊢ A
  → Γ ⊢ B
    -------------
  → Γ ⊢ C
_[_][_] {Γ} {A} {B} N V W = subst {Γ , A , B} {Γ} σ N
  where
  σ : ∀ {C} → Γ , A , B ∋ C → Γ ⊢ C
  σ Z         = W
```

```
σ (S Z)     = V
σ (S (S x)) = ` x
```

Values

```
data Value : ∀ {Γ A} → Γ ⊢ A → Set where

  -- functions

  V-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B}
      ---------------------------
    → Value (ƛ N)

  -- naturals

  V-zero : ∀ {Γ}
      -----------------
    → Value (`zero {Γ})

  V-suc_ : ∀ {Γ} {V : Γ ⊢ `ℕ}
    → Value V
      --------------
    → Value (`suc V)

  -- primitives

  V-con : ∀ {Γ n}
      -----------------
    → Value (con {Γ} n)

  -- products

  V-⟨_,_⟩ : ∀ {Γ A B} {V : Γ ⊢ A} {W : Γ ⊢ B}
    → Value V
    → Value W
      ----------------
    → Value `⟨ V , W ⟩
```

Implicit arguments need to be supplied when they are not fixed by the given arguments.

Reduction

```
infix 2 _⟶_

data _⟶_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  -- functions

  ξ-·₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A ⇒ B} {M : Γ ⊢ A}
    → L ⟶ L′
      ---------------
    → L · M ⟶ L′ · M

  ξ-·₂ : ∀ {Γ A B} {V : Γ ⊢ A ⇒ B} {M M′ : Γ ⊢ A}
```

```
  → Value V
  → M ⟶ M′
    ---------------
  → V · M ⟶ V · M′

β-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B} {V : Γ ⊢ A}
  → Value V
    --------------------
  → (ƛ N) · V ⟶ N [ V ]

-- naturals

ξ-suc : ∀ {Γ} {M M′ : Γ ⊢ `ℕ}
  → M ⟶ M′
    -----------------
  → `suc M ⟶ `suc M′

ξ-case : ∀ {Γ A} {L L′ : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
  → L ⟶ L′
    ------------------------
  → case L M N ⟶ case L′ M N

β-zero : ∀ {Γ A} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
    -------------------
  → case `zero M N ⟶ M

β-suc : ∀ {Γ A} {V : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
  → Value V
    ----------------------------
  → case (`suc V) M N ⟶ N [ V ]

-- fixpoint

β-μ : ∀ {Γ A} {N : Γ , A ⊢ A}
    ----------------
  → μ N ⟶ N [ μ N ]

-- primitive numbers

ξ-*₁ : ∀ {Γ} {L L′ M : Γ ⊢ Nat}
  → L ⟶ L′
    -----------------
  → L `* M ⟶ L′ `* M

ξ-*₂ : ∀ {Γ} {V M M′ : Γ ⊢ Nat}
  → Value V
  → M ⟶ M′
    -----------------
  → V `* M ⟶ V `* M′

δ-* : ∀ {Γ c d}
    ---------------------------------
  → con {Γ} c `* con d ⟶ con (c * d)

-- let

ξ-let : ∀ {Γ A B} {M M′ : Γ ⊢ A} {N : Γ , A ⊢ B}
  → M ⟶ M′
    ---------------------
  → `let M N ⟶ `let M′ N

β-let : ∀ {Γ A B} {V : Γ ⊢ A} {N : Γ , A ⊢ B}
  → Value V
```

```
    ------------------
  → `let V N ⟶ N [ V ]

-- products

ξ-⟨,⟩₁ : ∀ {Γ A B} {M M′ : Γ ⊢ A} {N : Γ ⊢ B}
  → M ⟶ M′
    -------------------------
  → `⟨ M , N ⟩ ⟶ `⟨ M′ , N ⟩

ξ-⟨,⟩₂ : ∀ {Γ A B} {V : Γ ⊢ A} {N N′ : Γ ⊢ B}
  → Value V
  → N ⟶ N′
    -------------------------
  → `⟨ V , N ⟩ ⟶ `⟨ V , N′ ⟩

ξ-proj₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A `× B}
  → L ⟶ L′
    --------------------
  → `proj₁ L ⟶ `proj₁ L′

ξ-proj₂ : ∀ {Γ A B} {L L′ : Γ ⊢ A `× B}
  → L ⟶ L′
    --------------------
  → `proj₂ L ⟶ `proj₂ L′

β-proj₁ : ∀ {Γ A B} {V : Γ ⊢ A} {W : Γ ⊢ B}
  → Value V
  → Value W
    ----------------------
  → `proj₁ `⟨ V , W ⟩ ⟶ V

β-proj₂ : ∀ {Γ A B} {V : Γ ⊢ A} {W : Γ ⊢ B}
  → Value V
  → Value W
    ----------------------
  → `proj₂ `⟨ V , W ⟩ ⟶ W

-- alternative formulation of products

ξ-case× : ∀ {Γ A B C} {L L′ : Γ ⊢ A `× B} {M : Γ , A , B ⊢ C}
  → L ⟶ L′
    ----------------------
  → case× L M ⟶ case× L′ M

β-case× : ∀ {Γ A B C} {V : Γ ⊢ A} {W : Γ ⊢ B} {M : Γ , A , B ⊢ C}
  → Value V
  → Value W
    ---------------------------------
  → case× `⟨ V , W ⟩ M ⟶ M [ V ][ W ]
```

Reflexive and transitive closure

```
infix 2 _—↠_
infix 1 begin_
infixr 2 _—→⟨_⟩_
infix 3 _∎

data _—↠_ {Γ A} : (Γ ⊢ A) → (Γ ⊢ A) → Set where

  _∎ : (M : Γ ⊢ A)
     ------
    → M —↠ M

  _—→⟨_⟩_ : (L : Γ ⊢ A) {M N : Γ ⊢ A}
    → L —→ M
    → M —↠ N
      ------
    → L —↠ N

begin_ : ∀ {Γ A} {M N : Γ ⊢ A}
  → M —↠ N
    ------
  → M —↠ N
begin M—↠N = M—↠N
```

Values do not reduce

```
V¬—→ : ∀ {Γ A} {M N : Γ ⊢ A}
  → Value M
    ----------
  → ¬ (M —→ N)
V¬—→ V-ƛ        ()
V¬—→ V-zero     ()
V¬—→ (V-suc VM) (ξ-suc M—→M′)     = V¬—→ VM M—→M′
V¬—→ V-con      ()
V¬—→ V-⟨ VM , _ ⟩ (ξ-⟨,⟩₁ M—→M′)  = V¬—→ VM M—→M′
V¬—→ V-⟨ _ , VN ⟩ (ξ-⟨,⟩₂ _ N—→N′) = V¬—→ VN N—→N′
```

Progress

```
data Progress {A} (M : ∅ ⊢ A) : Set where
  step : ∀ {N : ∅ ⊢ A}
    → M —→ N
      ----------
    → Progress M

  done :
      Value M
      ----------
    → Progress M
```

```
progress : ∀ {A}
  → (M : ∅ ⊢ A)
    -----------
  → Progress M
progress (` ())
progress (ƛ N)                 = done V-ƛ
progress (L · M) with progress L
... | step L⟶L′                = step (ξ-·₁ L⟶L′)
... | done V-ƛ with progress M
... | step M⟶M′                = step (ξ-·₂ V-ƛ M⟶M′)
... | done VM                  = step (β-ƛ VM)
progress (`zero)               = done V-zero
progress (`suc M) with progress M
... | step M⟶M′                = step (ξ-suc M⟶M′)
... | done VM                  = done (V-suc VM)
progress (case L M N) with progress L
... | step L⟶L′                = step (ξ-case L⟶L′)
... | done V-zero              = step β-zero
... | done (V-suc VL)          = step (β-suc VL)
progress (μ N)                 = step β-μ
progress (con n)               = done V-con
progress (L `* M) with progress L
... | step L⟶L′                = step (ξ-*₁ L⟶L′)
... | done V-con with progress M
... | step M⟶M′                = step (ξ-*₂ V-con M⟶M′)
... | done V-con               = step δ-*
progress (`let M N) with progress M
... | step M⟶M′                = step (ξ-let M⟶M′)
... | done VM                  = step (β-let VM)
progress `⟨ M , N ⟩ with progress M
... | step M⟶M′                = step (ξ-⟨,⟩₁ M⟶M′)
... | done VM with progress N
... | step N⟶N′                = step (ξ-⟨,⟩₂ VM N⟶N′)
... | done VN                  = done (V-⟨ VM , VN ⟩)
progress (`proj₁ L) with progress L
... | step L⟶L′                = step (ξ-proj₁ L⟶L′)
... | done (V-⟨ VM , VN ⟩)     = step (β-proj₁ VM VN)
progress (`proj₂ L) with progress L
... | step L⟶L′                = step (ξ-proj₂ L⟶L′)
... | done (V-⟨ VM , VN ⟩)     = step (β-proj₂ VM VN)
progress (case× L M) with progress L
... | step L⟶L′                = step (ξ-case× L⟶L′)
... | done (V-⟨ VM , VN ⟩)     = step (β-case× VM VN)
```

Evaluation

```
record Gas : Set where
  constructor gas
  field
    amount : ℕ

data Finished {Γ A} (N : Γ ⊢ A) : Set where

  done :
    Value N
    ----------
```

```
    → Finished N

  out-of-gas :
    ----------
    Finished N

data Steps {A} : ∅ ⊢ A → Set where

  steps : {L N : ∅ ⊢ A}
    → L —↠ N
    → Finished N
      ----------
    → Steps L

eval : ∀ {A}
  → Gas
  → (L : ∅ ⊢ A)
    -----------
  → Steps L
eval (gas zero) L    = steps (L ∎) out-of-gas
eval (gas (suc m)) L with progress L
... | done VL        = steps (L ∎) (done VL)
... | step {M} L—→M with eval (gas m) M
... | steps M—↠N fin = steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

Examples

```
cube : ∅ ⊢ Nat ⇒ Nat
cube = ƛ (# 0 `* # 0 `* # 0)

_ : cube · con 2 —↠ con 8
_ =
  begin
    cube · con 2
  —→⟨ β-ƛ V-con ⟩
    con 2 `* con 2 `* con 2
  —→⟨ ξ-*₁ δ-* ⟩
    con 4 `* con 2
  —→⟨ δ-* ⟩
    con 8
  ∎

exp10 : ∅ ⊢ Nat ⇒ Nat
exp10 = ƛ (`let (# 0 `* # 0)
           (`let (# 0 `* # 0)
             (`let (# 0 `* # 2)
               (# 0 `* # 0))))

_ : exp10 · con 2 —↠ con 1024
_ =
  begin
    exp10 · con 2
  —→⟨ β-ƛ V-con ⟩
    `let (con 2 `* con 2) (`let (# 0 `* # 0) (`let (# 0 `* con 2) (# 0 `* # 0)))
  —→⟨ ξ-let δ-* ⟩
    `let (con 4) (`let (# 0 `* # 0) (`let (# 0 `* con 2) (# 0 `* # 0)))
  —→⟨ β-let V-con ⟩
```

```
    `let (con 4 `* con 4) (`let (# 0 `* con 2) (# 0 `* # 0))
  ⟶⟨ ξ-let δ-* ⟩
    `let (con 16) (`let (# 0 `* con 2) (# 0 `* # 0))
  ⟶⟨ β-let V-con ⟩
    `let (con 16 `* con 2) (# 0 `* # 0)
  ⟶⟨ ξ-let δ-* ⟩
    `let (con 32) (# 0 `* # 0)
  ⟶⟨ β-let V-con ⟩
    con 32 `* con 32
  ⟶⟨ δ-* ⟩
    con 1024
  ∎

swap× : ∀ {A B} → ∅ ⊢ A `× B ⇒ B `× A
swap× = ƛ `⟨ `proj₂ (# 0) , `proj₁ (# 0) ⟩

_ : swap× · `⟨ con 42 , `zero ⟩ —↠ `⟨ `zero , con 42 ⟩
_ =
  begin
    swap× · `⟨ con 42 , `zero ⟩
  ⟶⟨ β-ƛ V-⟨ V-con , V-zero ⟩ ⟩
    `⟨ `proj₂ `⟨ con 42 , `zero ⟩ , `proj₁ `⟨ con 42 , `zero ⟩ ⟩
  ⟶⟨ ξ-⟨,⟩₁ (β-proj₂ V-con V-zero) ⟩
    `⟨ `zero , `proj₁ `⟨ con 42 , `zero ⟩ ⟩
  ⟶⟨ ξ-⟨,⟩₂ V-zero (β-proj₁ V-con V-zero) ⟩
    `⟨ `zero , con 42 ⟩
  ∎

swap×-case : ∀ {A B} → ∅ ⊢ A `× B ⇒ B `× A
swap×-case = ƛ case× (# 0) `⟨ # 0 , # 1 ⟩

_ : swap×-case · `⟨ con 42 , `zero ⟩ —↠ `⟨ `zero , con 42 ⟩
_ =
  begin
     swap×-case · `⟨ con 42 , `zero ⟩
    ⟶⟨ β-ƛ V-⟨ V-con , V-zero ⟩ ⟩
      case× `⟨ con 42 , `zero ⟩ `⟨ # 0 , # 1 ⟩
    ⟶⟨ β-case× V-con V-zero ⟩
      `⟨ `zero , con 42 ⟩
    ∎
```

Exercise `More` (recommended and practice)

Formalise the remaining constructs defined in this chapter. Make your changes in this file. Evaluate each example, applied to data as needed, to confirm it returns the expected answer:

- sums (recommended)
- unit type (practice)
- an alternative formulation of unit type (practice)
- empty type (recommended)
- lists (practice)

Please delimit any code you add as follows:

```
-- begin
-- end
```

Exercise `double-subst` (stretch)

Show that a double substitution is equivalent to two single substitutions.

```
postulate
  double-subst :
    ∀ {Γ A B C} {V : Γ ⊢ A} {W : Γ ⊢ B} {N : Γ , A , B ⊢ C} →
      N [ V ][ W ] ≡ (N [ rename S_ W ]) [ V ]
```

Note the arguments need to be swapped and `W` needs to have its context adjusted via renaming in order for the right-hand side to be well typed.

Test examples

We repeat the test examples from Chapter DeBruijn, in order to make sure we have not broken anything in the process of extending our base calculus.

```
two : ∀ {Γ} → Γ ⊢ `ℕ
two = `suc `suc `zero

plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = μ ƛ ƛ (case (# 1) (# 0) (`suc (# 3 · # 0 · # 1)))

2+2 : ∀ {Γ} → Γ ⊢ `ℕ
2+2 = plus · two · two

Ch : Type → Type
Ch A = (A ⇒ A) ⇒ A ⇒ A

twoᶜ : ∀ {Γ A} → Γ ⊢ Ch A
twoᶜ = ƛ ƛ (# 1 · (# 1 · # 0))

plusᶜ : ∀ {Γ A} → Γ ⊢ Ch A ⇒ Ch A ⇒ Ch A
plusᶜ = ƛ ƛ ƛ ƛ (# 3 · # 1 · (# 2 · # 1 · # 0))

sucᶜ : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ
sucᶜ = ƛ `suc (# 0)

2+2ᶜ : ∀ {Γ} → Γ ⊢ `ℕ
2+2ᶜ = plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

Unicode

This chapter uses the following unicode:

```
σ  U+03C3  GREEK SMALL LETTER SIGMA (\Gs or \sigma)
†  U+2020  DAGGER (\dag)
‡  U+2021  DOUBLE DAGGER (\ddag)
```

Chapter 15

Bisimulation: Relating reduction systems

```
module plfa.part2.Bisimulation where
```

Some constructs can be defined in terms of other constructs. In the previous chapter, we saw how *let* terms can be rewritten as an application of an abstraction, and how two alternative formulations of products — one with projections and one with case — can be formulated in terms of each other. In this chapter, we look at how to formalise such claims.

Given two different systems, with different terms and reduction rules, we define what it means to claim that one *simulates* the other. Let's call our two systems *source* and *target*. Let `M`, `N` range over terms of the source, and `M†`, `N†` range over terms of the target. We define a relation

```
M ~ M†
```

between corresponding terms of the two systems. We have a *simulation* of the source by the target if every reduction in the source has a corresponding reduction sequence in the target:

Simulation: For every `M`, `M†`, and `N`: If `M ~ M†` and `M —→ N` then `M† —↠ N†` and `N ~ N†` for some `N†`.

Or, in a diagram:

```
M  --- —→ --- N
|             |
|             |
~             ~
|             |
|             |
M† --- —↠ --- N†
```

Sometimes we will have a stronger condition, where each reduction in the source corresponds to a reduction (rather than a reduction sequence) in the target:

```
M  --- —→ --- N
|             |
|             |
~             ~
|             |
|             |
```

```
M† --- ⟶ --- N†
```

This stronger condition is known as *lock-step* or *on the nose* simulation.

We are particularly interested in the situation where there is also a simulation from the target to the source: every reduction in the target has a corresponding reduction sequence in the source. This situation is called a *bisimulation*.

Simulation is established by case analysis over all possible reductions and all possible terms to which they are related. For each reduction step in the source we must show a corresponding reduction sequence in the target.

For instance, the source might be lambda calculus with *let* added, and the target the same system with `let` translated out. The key rule defining our relation will be:

```
M ~ M†
N ~ N†
-------------------------------
let x = M in N ~ (ƛ x ⇒ N†) · M†
```

All the other rules are congruences: variables relate to themselves, and abstractions and applications relate if their components relate:

```
-----
x ~ x

N ~ N†
------------------
ƛ x ⇒ N ~ ƛ x ⇒ N†

L ~ L†
M ~ M†
---------------
L · M ~ L† · M†
```

Covering the other constructs of our language — naturals, fixpoints, products, and so on — would add little save length.

In this case, our relation can be specified by a function from source to target:

```
(x) †               =  x
(ƛ x ⇒ N) †         =  ƛ x ⇒ (N †)
(L · M) †           =  (L †) · (M †)
(let x = M in N) †  =  (ƛ x ⇒ (N †)) · (M †)
```

And we have

```
M † ≡ N
-------
M ~ N
```

and conversely. But in general we may have a relation without any corresponding function.

This chapter formalises establishing that `~` as defined above is a simulation from source to target. We leave establishing it in the reverse direction as an exercise. Another exercise is to show the alternative formulations of products in Chapter More are in bisimulation.

Imports

We import our source language from Chapter More:

```
open import plfa.part2.More
```

Simulation

The simulation is a straightforward formalisation of the rules in the introduction:

```
infix 4 _~_
infix 5 ~ƛ_
infix 7 _~·_

data _~_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  ~` : ∀ {Γ A} {x : Γ ∋ A}
    ---------
   → ` x ~ ` x

  ~ƛ_ : ∀ {Γ A B} {N N† : Γ , A ⊢ B}
   → N ~ N†
     ----------
   → ƛ N ~ ƛ N†

  _~·_ : ∀ {Γ A B} {L L† : Γ ⊢ A ⇒ B} {M M† : Γ ⊢ A}
   → L ~ L†
   → M ~ M†
     ---------------
   → L · M ~ L† · M†

  ~let : ∀ {Γ A B} {M M† : Γ ⊢ A} {N N† : Γ , A ⊢ B}
   → M ~ M†
   → N ~ N†
     ----------------------
   → `let M N ~ (ƛ N†) · M†
```

The language in Chapter More has more constructs, which we could easily add. However, leaving the simulation small lets us focus on the essence. It's a handy technical trick that we can have a large source language, but only bother to include in the simulation the terms of interest.

Exercise `_†` (practice)

Formalise the translation from source to target given in the introduction. Show that `M † ≡ N` implies `M ~ N`, and conversely.

Hint: For simplicity, we focus on only a few constructs of the language, so `_†` should be defined only on relevant terms. One way to do this is to use a decidable predicate to pick out terms in the domain of `_†`, using proof by reflection.

```
-- Your code goes here
```

Simulation commutes with values

We need a number of technical results. The first is that simulation commutes with values. That is, if `M ~ M†` and `M` is a value then `M†` is also a value:

```
~val : ∀ {Γ A} {M M† : Γ ⊢ A}
  → M ~ M†
  → Value M
    --------
  → Value M†
~val ~`          ()
~val (~ƛ ~N)     V-ƛ = V-ƛ
~val (~L ~· ~M) ()
~val (~let ~M ~N) ()
```

It is a straightforward case analysis, where here the only value of interest is a lambda abstraction.

Exercise `~val⁻¹` (practice)

Show that this also holds in the reverse direction: if `M ~ M†` and `Value M†` then `Value M`.

```
-- Your code goes here
```

Simulation commutes with renaming

The next technical result is that simulation commutes with renaming. That is, if `ρ` maps any judgment `Γ ∋ A` to a judgment `Δ ∋ A`, and if `M ~ M†` then `rename ρ M ~ rename ρ M†`:

```
~rename : ∀ {Γ Δ}
  → (ρ : ∀ {A} → Γ ∋ A → Δ ∋ A)
    ------------------------------------------------------
  → (∀ {A} {M M† : Γ ⊢ A} → M ~ M† → rename ρ M ~ rename ρ M†)
~rename ρ (~`)          = ~`
~rename ρ (~ƛ ~N)       = ~ƛ (~rename (ext ρ) ~N)
~rename ρ (~L ~· ~M)    = (~rename ρ ~L) ~· (~rename ρ ~M)
~rename ρ (~let ~M ~N) = ~let (~rename ρ ~M) (~rename (ext ρ) ~N)
```

The structure of the proof is similar to the structure of renaming itself: reconstruct each term with recursive invocation, extending the environment where appropriate (in this case, only for the body of an abstraction).

Simulation commutes with substitution

The third technical result is that simulation commutes with substitution. It is more complex than renaming, because where we had one renaming map `ρ` here we need two substitution maps, `σ` and `σ†`.

The proof first requires we establish an analogue of extension. If `σ` and `σ†` both map any judgment `Γ ∋ A` to a judgment `Δ ⊢ A`, such that for every `x` in `Γ ∋ A` we have `σ x ~ σ† x`, then

for any `x` in `Γ , B ∋ A` we have `exts σ x ~ exts σ† x`:

```
~exts : ∀ {Γ Δ}
  → {σ : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → {σ† : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → (∀ {A} → (x : Γ ∋ A) → σ x ~ σ† x)
    ------------------------------------------------
  → (∀ {A B} → (x : Γ , B ∋ A) → exts σ x ~ exts σ† x)
~exts ~σ Z     = ~`
~exts ~σ (S x) = ~rename S_ (~σ x)
```

The structure of the proof is similar to the structure of extension itself. The newly introduced variable trivially relates to itself, and otherwise we apply renaming to the hypothesis.

With extension under our belts, it is straightforward to show substitution commutes. If `σ` and `σ†` both map any judgment `Γ ∋ A` to a judgment `Δ ⊢ A`, such that for every `x` in `Γ ∋ A` we have `σ x ~ σ† x`, and if `M ~ M†`, then `subst σ M ~ subst σ† M†`:

```
~subst : ∀ {Γ Δ}
  → {σ : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → {σ† : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → (∀ {A} → (x : Γ ∋ A) → σ x ~ σ† x)
    ------------------------------------------------------
  → (∀ {A} {M M† : Γ ⊢ A} → M ~ M† → subst σ M ~ subst σ† M†)
~subst ~σ (~` {x = x}) = ~σ x
~subst ~σ (~ƛ ~N)      = ~ƛ (~subst (~exts ~σ) ~N)
~subst ~σ (~L ~· ~M)   = (~subst ~σ ~L) ~· (~subst ~σ ~M)
~subst ~σ (~let ~M ~N) = ~let (~subst ~σ ~M) (~subst (~exts ~σ) ~N)
```

Again, the structure of the proof is similar to the structure of substitution itself: reconstruct each term with recursive invocation, extending the environment where appropriate (in this case, only for the body of an abstraction).

From the general case of substitution, it is also easy to derive the required special case. If `N ~ N†` and `M ~ M†`, then `N [ M ] ~ N† [ M† ]`:

```
~sub : ∀ {Γ A B} {N N† : Γ , B ⊢ A} {M M† : Γ ⊢ B}
  → N ~ N†
  → M ~ M†
    -----------------------
  → (N [ M ]) ~ (N† [ M† ])
~sub {Γ} {A} {B} ~N ~M = ~subst {Γ , B} {Γ} ~σ {A} ~N
  where
  ~σ : ∀ {A} → (x : Γ , B ∋ A) → _ ~ _
  ~σ Z     = ~M
  ~σ (S x) = ~`
```

Once more, the structure of the proof resembles the original.

The relation is a simulation

Finally, we can show that the relation actually is a simulation. In fact, we will show the stronger condition of a lock-step simulation. What we wish to show is:

Lock-step simulation: For every `M`, `M†`, and `N`: If `M ~ M†` and `M ⟶ N` then `M† ⟶ N†` and `N ~ N†` for some `N†`.

Or, in a diagram:

```
M  --- ⟶ --- N
|             |
|             |
~             ~
|             |
|             |
M† --- ⟶ --- N†
```

We first formulate a concept corresponding to the lower leg of the diagram, that is, its right and bottom edges:

```
data Leg {Γ A} (M† N : Γ ⊢ A) : Set where

  leg : ∀ {N† : Γ ⊢ A}
    → N ~ N†
    → M† ⟶ N†
      --------
    → Leg M† N
```

For our formalisation, in this case, we can use a stronger relation than `⟶↠`, replacing it by `⟶`.

We can now state and prove that the relation is a simulation. Again, in this case, we can use a stronger relation than `⟶↠`, replacing it by `⟶`:

```
sim : ∀ {Γ A} {M M† N : Γ ⊢ A}
  → M ~ M†
  → M ⟶ N
    ---------
  → Leg M† N
sim ~` ()
sim (~ƛ ~N) ()
sim (~L ~· ~M) (ξ-·₁ L⟶)
  with sim ~L L⟶
... | leg ~L′ L†⟶ = leg (~L′ ~· ~M) (ξ-·₁ L†⟶)
sim (~V ~· ~M) (ξ-·₂ VV M⟶)
  with sim ~M M⟶
... | leg ~M′ M†⟶ = leg (~V ~· ~M′) (ξ-·₂ (~val ~V VV) M†⟶)
sim ((~ƛ ~N) ~· ~V) (β-ƛ VV) = leg (~sub ~N ~V) (β-ƛ (~val ~V VV))
sim (~let ~M ~N) (ξ-let M⟶)
  with sim ~M M⟶
... | leg ~M′ M†⟶ = leg (~let ~M′ ~N) (ξ-·₂ V-ƛ M†⟶)
sim (~let ~V ~N) (β-let VV) = leg (~sub ~N ~V) (β-ƛ (~val ~V VV))
```

The proof is by case analysis, examining each possible instance of `M ~ M†` and each possible instance of `M ⟶ M†`, using recursive invocation whenever the reduction is by a `ξ` rule, and hence contains another reduction. In its structure, it looks a little bit like a proof of progress:

- If the related terms are variables, no reduction applies.
- If the related terms are abstractions, no reduction applies.
- If the related terms are applications, there are three subcases:
  - The source term reduces via `ξ-·₁`, in which case the target term does as well. Recursive invocation gives us

```
L  --- ⟶ ---  L′
|             |
|             |
~             ~
|             |
|             |
L† --- ⟶ --- L′†
```

from which follows:

```
 L · M  --- ⟶ ---  L′ · M
   |                  |
   |                  |
   ~                  ~
   |                  |
   |                  |
L† · M† --- ⟶ --- L′† · M†
```

- The source term reduces via `ξ-·₂`, in which case the target term does as well. Recursive invocation gives us

```
M  --- ⟶ ---  M′
|             |
|             |
~             ~
|             |
|             |
M† --- ⟶ --- M′†
```

from which follows:

```
 V · M  --- ⟶ ---  V · M′
   |                  |
   |                  |
   ~                  ~
   |                  |
   |                  |
V† · M† --- ⟶ --- V† · M′†
```

Since simulation commutes with values and `V` is a value, `V†` is also a value.

- The source term reduces via `β-ƛ`, in which case the target term does as well:

```
 (ƛ x ⇒ N) · V  --- ⟶ ---  N [ x := V ]
      |                          |
      |                          |
      ~                          ~
      |                          |
      |                          |
(ƛ x ⇒ N†) · V† --- ⟶ --- N† [ x :=  V† ]
```

Since simulation commutes with values and `V` is a value, `V†` is also a value. Since simulation commutes with substitution and `N ~ N†` and `V ~ V†`, we have `N [ x := V ] ~ N† [ x := V† ]`.

- If the related terms are a let and an application of an abstraction, there are two subcases:
  - The source term reduces via `ξ-let`, in which case the target term reduces via `ξ-·₂`. Recursive invocation gives us

```
M  --- ⟶ ---  M′
|              |
|              |
~              ~
|              |
|              |
M† --- ⟶ --- M′†
```

from which follows:

```
let x = M in N --- ⟶ --- let x = M′ in N
      |                          |
      |                          |
      ~                          ~
      |                          |
      |                          |
(ƛ x ⇒ N) · M  --- ⟶ --- (ƛ x ⇒ N) · M′
```

- The source term reduces via `β-let`, in which case the target term reduces via `β-ƛ`:

```
let x = V in N  --- ⟶ ---  N [ x := V ]
      |                          |
      |                          |
      ~                          ~
      |                          |
      |                          |
(ƛ x ⇒ N†) · V† --- ⟶ --- N† [ x := V† ]
```

Since simulation commutes with values and `V` is a value, `V†` is also a value. Since simulation commutes with substitution and `N ~ N†` and `V ~ V†`, we have `N [ x := V ] ~ N† [ x := V† ]`.

Exercise `sim⁻¹` (practice)

Show that we also have a simulation in the other direction, and hence that we have a bisimulation.

```
-- Your code goes here
```

Exercise `products` (practice)

Show that the two formulations of products in Chapter More are in bisimulation. The only constructs you need to include are variables, and those connected to functions and products. In this case, the simulation is *not* lock-step.

```
-- Your code goes here
```

Unicode

This chapter uses the following unicode:

```
†  U+2020  DAGGER (\dag)
⁻  U+207B  SUPERSCRIPT MINUS (\^-)
¹  U+00B9  SUPERSCRIPT ONE (\^1)
```

Chapter 16

Inference: Bidirectional type inference

```
module plfa.part2.Inference where
```

So far in our development, type derivations for the corresponding term have been provided by fiat. In Chapter Lambda type derivations are extrinsic to the term, while in Chapter DeBruijn type derivations are intrinsic to the term, but in both we have written out the type derivations in full.

In practice, one often writes down a term with a few decorations and applies an algorithm to *infer* the corresponding type derivation. Indeed, this is exactly what happens in Agda: we specify the types for top-level function declarations, and type information for everything else is inferred from what has been given. The style of inference Agda uses is based on a technique called *bidirectional* type inference, which will be presented in this chapter.

This chapter ties our previous developments together. We begin with a term with some type annotations, close to the raw terms of Chapter Lambda, and from it we compute an intrinsically-typed term, in the style of Chapter DeBruijn.

Introduction: Inference rules as algorithms

In the calculus we have considered so far, a term may have more than one type. For example,

```
(ƛ x ⇒ x) ∶ (A ⇒ A)
```

holds for *every* type `A`. We start by considering a small language for lambda terms where every term has a unique type. All we need do is decorate each abstraction term with the type of its argument. This gives us the grammar:

```
L, M, N ::=                       decorated terms
  x                                 variable
  ƛ x ∶ A ⇒ N                       abstraction (decorated)
  L · M                             application
```

Each of the associated type rules can be read as an algorithm for type checking. For each typing judgment, we label each position as either an *input* or an *output*.

For the judgment

```
Γ ∋ x ⦂ A
```

we take the context `Γ` and the variable `x` as inputs, and the type `A` as output. Consider the rules:

```
---------------- Z
Γ , x ⦂ A ∋ x ⦂ A

Γ ∋ x ⦂ A
---------------- S
Γ , y ⦂ B ∋ x ⦂ A
```

From the inputs we can determine which rule applies: if the last variable in the context matches the given variable then the first rule applies, else the second. (For de Bruijn indices, it is even easier: zero matches the first rule and successor the second.) For the first rule, the output type can be read off as the last type in the input context. For the second rule, the inputs of the conclusion determine the inputs of the hypothesis, and the output of the hypothesis determines the output of the conclusion.

For the judgment

```
Γ ⊢ M ⦂ A
```

we take the context `Γ` and term `M` as inputs, and the type `A` as output. Consider the rules:

```
Γ ∋ x ⦂ A
-----------
Γ ⊢ ` x ⦂ A

Γ , x ⦂ A ⊢ N ⦂ B
--------------------------
Γ ⊢ (ƛ x ⦂ A ⇒ N) ⦂ (A ⇒ B)

Γ ⊢ L ⦂ A ⇒ B
Γ ⊢ M ⦂ A′
A ≡ A′
-------------
Γ ⊢ L · M ⦂ B
```

The input term determines which rule applies: variables use the first rule, abstractions the second, and applications the third. We say such rules are *syntax directed*. For the variable rule, the inputs of the conclusion determine the inputs of the hypothesis, and the output of the hypothesis determines the output of the conclusion. Same for the abstraction rule — the bound variable and argument are carried from the term of the conclusion into the context of the hypothesis; this works because we added the argument type to the abstraction. For the application rule, we add a third hypothesis to check whether the domain of the function matches the type of the argument; this judgment is decidable when both types are given as inputs. The inputs of the conclusion determine the inputs of the first two hypotheses, the outputs of the first two hypotheses determine the inputs of the third hypothesis, and the output of the first hypothesis determines the output of the conclusion.

Converting the above to an algorithm is straightforward, as is adding naturals and fixpoint. We omit the details. Instead, we consider a detailed description of an approach that requires less obtrusive decoration. The idea is to break the normal typing judgment into two judgments, one that produces the type as an output (as above), and another that takes it as an input.

Synthesising and inheriting types

In addition to the lookup judgment for variables, which will remain as before, we now have two judgments for the type of the term:

```
Γ ⊢ M ↑ A
Γ ⊢ M ↓ A
```

The first of these *synthesises* the type of a term, as before, while the second *inherits* the type. In the first, the context and term are inputs and the type is an output; while in the second, all three of the context, term, and type are inputs.

Which terms use synthesis and which inheritance? Our approach will be that the main term in a *deconstructor* is typed via synthesis while *constructors* are typed via inheritance. For instance, the function in an application is typed via synthesis, but an abstraction is typed via inheritance. The inherited type in an abstraction term serves the same purpose as the argument type decoration of the previous section.

Terms that deconstruct a value of a type always have a main term (supplying an argument of the required type) and often have side-terms. For application, the main term supplies the function and the side term supplies the argument. For case terms, the main term supplies a natural and the side terms are the two branches. In a deconstructor, the main term will be typed using synthesis but the side terms will be typed using inheritance. As we will see, this leads naturally to an application as a whole being typed by synthesis, while a case term as a whole will be typed by inheritance. Variables are naturally typed by synthesis, since we can look up the type in the input context. Fixed points will be naturally typed by inheritance.

In order to get a syntax-directed type system we break terms into two kinds, `Term⁺` and `Term⁻`, which are typed by synthesis and inheritance, respectively. A subterm that is typed by synthesis may appear in a context where it is typed by inheritance, or vice-versa, and this gives rise to two new term forms.

For instance, we said above that the argument of an application is typed by inheritance and that variables are typed by synthesis, giving a mismatch if the argument of an application is a variable. Hence, we need a way to treat a synthesized term as if it is inherited. We introduce a new term form, `M ↑` for this purpose. The typing judgment checks that the inherited and synthesised types match.

Similarly, we said above that the function of an application is typed by synthesis and that abstractions are typed by inheritance, giving a mismatch if the function of an application is an abstraction. Hence, we need a way to treat an inherited term as if it is synthesised. We introduce a new term form `M ↓ A` for this purpose. The typing judgment returns `A` as the synthesized type of the term as a whole, as well as using it as the inherited type for `M`.

The term form `M ↓ A` represents the only place terms need to be decorated with types. It only appears when switching from synthesis to inheritance, that is, when a term that *deconstructs* a value of a type contains as its main term a term that *constructs* a value of a type, in other words, a place where a `β`-reduction will occur. Typically, we will find that decorations are only required on top level declarations.

We can extract the grammar for terms from the above:

```
L⁺, M⁺, N⁺ ::=                  terms with synthesized type
  x                               variable
  L⁺ · M⁻                         application
  M⁻ ↓ A                          switch to inherited

L⁻, M⁻, N⁻ ::=                  terms with inherited type
  ƛ x ⇒ N⁻                        abstraction
```

```
  `zero                                zero
  `suc M⁻                              successor
  case L⁺ [zero⇒ M⁻ |suc x ⇒ N⁻ ]      case
  μ x ⇒ N⁻                             fixpoint
  M⁺ ↑                                 switch to synthesized
```

We will formalise the above shortly.

Soundness and completeness

What we intend to show is that the typing judgments are *decidable*:

```
synthesize : ∀ (Γ : Context) (M : Term⁺)
    ----------------------
  → Dec (∃[ A ]( Γ ⊢ M ↑ A ))

inherit : ∀ (Γ : Context) (M : Term⁻) (A : Type)
          ---------------
        → Dec (Γ ⊢ M ↓ A)
```

Given context `Γ` and synthesised term `M`, we must decide whether there exists a type `A` such that `Γ ⊢ M ↑ A` holds, or its negation. Similarly, given context `Γ`, inherited term `M`, and type `A`, we must decide whether `Γ ⊢ M ↓ A` holds, or its negation.

Our proof is constructive. In the synthesised case, it will either deliver a pair of a type `A` and evidence that `Γ ⊢ M ↓ A`, or a function that given such a pair produces evidence of a contradiction. In the inherited case, it will either deliver evidence that `Γ ⊢ M ↑ A`, or a function that given such evidence produces evidence of a contradiction. The positive case is referred to as *soundness* — synthesis and inheritance succeed only if the corresponding relation holds. The negative case is referred to as *completeness* — synthesis and inheritance fail only when they cannot possibly succeed.

Another approach might be to return a derivation if synthesis or inheritance succeeds, and an error message otherwise — for instance, see the section of the Agda user manual discussing syntactic sugar. Such an approach demonstrates soundness, but not completeness. If it returns a derivation, we know it is correct; but there is nothing to prevent us from writing a function that *always* returns an error, even when there exists a correct derivation. Demonstrating both soundness and completeness is significantly stronger than demonstrating soundness alone. The negative proof can be thought of as a semantically verified error message, although in practice it may be less readable than a well-crafted error message.

We are now ready to begin the formal development.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; trans; cong; cong₂; _≢_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _+_; _*_)
open import Data.String using (String; _≟_)
open import Data.Product using (_×_; ∃; ∃-syntax) renaming (_,_ to ⟨_,_⟩)
open import Relation.Nullary using (¬_; Dec; yes; no)
```

Once we have a type derivation, it will be easy to construct from it the intrinsically-typed representation. In order that we can compare with our previous development, we import module `plfa.part2.More`:

```
import plfa.part2.More as DB
```

The phrase `as DB` allows us to refer to definitions from that module as, for instance, `DB._⊢_`, which is invoked as `Γ DB.⊢ A`, where `Γ` has type `DB.Context` and `A` has type `DB.Type`.

Syntax

First, we get all our infix declarations out of the way. We list separately operators for judgments and terms:

```
infix  4 _∋_⦂_
infix  4 _⊢_↑_
infix  4 _⊢_↓_
infixl 5 _,_⦂_

infixr 7 _⇒_

infix  5 ƛ_⇒_
infix  5 μ_⇒_
infix  6 _↑
infix  6 _↓_
infixl 7 _·_
infix  8 `suc_
infix  9 `_
```

Identifiers, types, and contexts are as before:

```
Id : Set
Id = String

data Type : Set where
  `ℕ  : Type
  _⇒_ : Type → Type → Type

data Context : Set where
  ∅   : Context
  _,_⦂_ : Context → Id → Type → Context
```

The syntax of terms is defined by mutual recursion. We use `Term⁺` and `Term⁻` for terms with synthesized and inherited types, respectively. Note the inclusion of the switching forms, `M ↓ A` and `M ↑`:

```
data Term⁺ : Set
data Term⁻ : Set

data Term⁺ where
  `_  : Id → Term⁺
  _·_ : Term⁺ → Term⁻ → Term⁺
  _↓_ : Term⁻ → Type → Term⁺

data Term⁻ where
  ƛ_⇒_                  : Id → Term⁻ → Term⁻
```

```
  `zero                : Term⁻
  `suc_                : Term⁻ → Term⁻
  `case_[zero⇒_|suc_⇒_] : Term⁺ → Term⁻ → Id → Term⁻ → Term⁻
  μ_⇒_                 : Id → Term⁻ → Term⁻
  _↑                   : Term⁺ → Term⁻
```

The choice as to whether each term is synthesized or inherited follows the discussion above, and can be read off from the informal grammar presented earlier. Main terms in deconstructors synthesise, constructors and side terms in deconstructors inherit.

Example terms

We can recreate the examples from preceding chapters. First, computing two plus two on naturals:

```
two : Term⁻
two = `suc (`suc `zero)

plus : Term⁺
plus = (μ "p" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
          `case (` "m") [zero⇒ ` "n" ↑
                        |suc "m" ⇒ `suc (` "p" · (` "m" ↑) · (` "n" ↑) ↑) ])
           ↓ (`ℕ ⇒ `ℕ ⇒ `ℕ)

2+2 : Term⁺
2+2 = plus · two · two
```

The only change is to decorate with down and up arrows as required. The only type decoration required is for `plus`.

Next, computing two plus two with Church numerals:

```
Ch : Type
Ch = (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ

twoᶜ : Term⁻
twoᶜ = (ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · (` "z" ↑) ↑) ↑)

plusᶜ : Term⁺
plusᶜ = (ƛ "m" ⇒ ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒
            ` "m" · (` "s" ↑) · (` "n" · (` "s" ↑) · (` "z" ↑) ↑) ↑)
             ↓ (Ch ⇒ Ch ⇒ Ch)

sucᶜ : Term⁻
sucᶜ = ƛ "x" ⇒ `suc (` "x" ↑)

2+2ᶜ : Term⁺
2+2ᶜ = plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

The only type decoration required is for `plusᶜ`. One is not even required for `sucᶜ`, which inherits its type as an argument of `plusᶜ`.

Bidirectional type checking

The typing rules for variables are as in Lambda:

```
data _∋_⦂_ : Context → Id → Type → Set where

  Z : ∀ {Γ x A}
    ---------------------
    → Γ , x ⦂ A ∋ x ⦂ A

  S : ∀ {Γ x y A B}
    → x ≢ y
    → Γ ∋ x ⦂ A
      ------------------
    → Γ , y ⦂ B ∋ x ⦂ A
```

As with syntax, the judgments for synthesizing and inheriting types are mutually recursive:

```
data _⊢_↑_ : Context → Term⁺ → Type → Set
data _⊢_↓_ : Context → Term⁻ → Type → Set

data _⊢_↑_ where

  ⊢` : ∀ {Γ A x}
    → Γ ∋ x ⦂ A
      -----------
    → Γ ⊢ ` x ↑ A

  _·_ : ∀ {Γ L M A B}
    → Γ ⊢ L ↑ A ⇒ B
    → Γ ⊢ M ↓ A
      -------------
    → Γ ⊢ L · M ↑ B

  ⊢↓ : ∀ {Γ M A}
    → Γ ⊢ M ↓ A
      ---------------
    → Γ ⊢ (M ↓ A) ↑ A

data _⊢_↓_ where

  ⊢ƛ : ∀ {Γ x N A B}
    → Γ , x ⦂ A ⊢ N ↓ B
      --------------------
    → Γ ⊢ ƛ x ⇒ N ↓ A ⇒ B

  ⊢zero : ∀ {Γ}
      --------------
    → Γ ⊢ `zero ↓ `ℕ

  ⊢suc : ∀ {Γ M}
    → Γ ⊢ M ↓ `ℕ
      ---------------
    → Γ ⊢ `suc M ↓ `ℕ

  ⊢case : ∀ {Γ L M x N A}
    → Γ ⊢ L ↑ `ℕ
    → Γ ⊢ M ↓ A
    → Γ , x ⦂ `ℕ ⊢ N ↓ A
      ------------------------------------
    → Γ ⊢ `case L [zero⇒ M |suc x ⇒ N ] ↓ A

  ⊢μ : ∀ {Γ x N A}
    → Γ , x ⦂ A ⊢ N ↓ A
      -----------------
```

```
    → Γ ⊢ μ x ⇒ N ↓ A

  ⊢↑ : ∀ {Γ M A B}
    → Γ ⊢ M ↑ A
    → A ≡ B
      -------------
    → Γ ⊢ (M ↑) ↓ B
```

We follow the same convention as Chapter Lambda, prefacing the constructor with `⊢` to derive the name of the corresponding type rule.

The rules are similar to those in Chapter Lambda, modified to support synthesised and inherited types. The two new rules are those for `⊢↓` and `⊢↑`. The former both passes the type decoration as the inherited type and returns it as the synthesised type. The latter takes the synthesised type and the inherited type and confirms they are identical — it should remind you of the equality test in the application rule in the first section.

Exercise `bidirectional-mul` (recommended)

Rewrite your definition of multiplication from Chapter Lambda, decorated to support inference.

```
-- Your code goes here
```

Exercise `bidirectional-products` (recommended)

Extend the bidirectional type rules to include products from Chapter More.

```
-- Your code goes here
```

Exercise `bidirectional-rest` (stretch)

Extend the bidirectional type rules to include the rest of the constructs from Chapter More.

```
-- Your code goes here
```

Prerequisites

The rule for `M ↑` requires the ability to decide whether two types are equal. It is straightforward to code:

```
_≟Tp_ : (A B : Type) → Dec (A ≡ B)
`ℕ ≟Tp `ℕ                = yes refl
`ℕ ≟Tp (A ⇒ B)           = no λ()
(A ⇒ B) ≟Tp `ℕ           = no λ()
(A ⇒ B) ≟Tp (A′ ⇒ B′)
  with A ≟Tp A′ | B ≟Tp B′
... | no A≢ | _          = no λ{refl → A≢ refl}
```

```
...  | yes _  | no B≠          = no λ{refl → B≠ refl}
...  | yes refl | yes refl = yes refl
```

We will also need a couple of obvious lemmas; the domain and range of equal function types are equal:

```
dom≡ : ∀ {A A′ B B′} → A ⇒ B ≡ A′ ⇒ B′ → A ≡ A′
dom≡ refl = refl

rng≡ : ∀ {A A′ B B′} → A ⇒ B ≡ A′ ⇒ B′ → B ≡ B′
rng≡ refl = refl
```

We will also need to know that the types `` `ℕ `` and `A ⇒ B` are not equal:

```
ℕ≠⇒ : ∀ {A B} → `ℕ ≠ A ⇒ B
ℕ≠⇒ ()
```

Unique types

Looking up a type in the context is unique. Given two derivations, one showing `Γ ∋ x ⦂ A` and one showing `Γ ∋ x ⦂ B`, it follows that `A` and `B` must be identical:

```
uniq-∋ : ∀ {Γ x A B} → Γ ∋ x ⦂ A → Γ ∋ x ⦂ B → A ≡ B
uniq-∋ Z Z                 = refl
uniq-∋ Z (S x≠y _)         = ⊥-elim (x≠y refl)
uniq-∋ (S x≠y _) Z         = ⊥-elim (x≠y refl)
uniq-∋ (S _ ∋x) (S _ ∋x′)  = uniq-∋ ∋x ∋x′
```

If both derivations are by rule `Z` then uniqueness follows immediately, while if both derivations are by rule `S` then uniqueness follows by induction. It is a contradiction if one derivation is by rule `Z` and one by rule `S`, since rule `Z` requires the variable we are looking for is the final one in the context, while rule `S` requires it is not.

Synthesizing a type is also unique. Given two derivations, one showing `Γ ⊢ M ↑ A` and one showing `Γ ⊢ M ↑ B`, it follows that `A` and `B` must be identical:

```
uniq-↑ : ∀ {Γ M A B} → Γ ⊢ M ↑ A → Γ ⊢ M ↑ B → A ≡ B
uniq-↑ (⊢` ∋x) (⊢` ∋x′)      = uniq-∋ ∋x ∋x′
uniq-↑ (⊢L · ⊢M) (⊢L′ · ⊢M′) = rng≡ (uniq-↑ ⊢L ⊢L′)
uniq-↑ (⊢↓ ⊢M) (⊢↓ ⊢M′)      = refl
```

There are three possibilities for the term. If it is a variable, uniqueness of synthesis follows from uniqueness of lookup. If it is an application, uniqueness follows by induction on the function in the application, since the range of equal types are equal. If it is a switch expression, uniqueness follows since both terms are decorated with the same type.

Lookup type of a variable in the context

Given `Γ` and two distinct variables `x` and `y`, if there is no type `A` such that `Γ ∋ x ⦂ A` holds, then there is also no type `A` such that `Γ , y ⦂ B ∋ x ⦂ A` holds:

```
extϑ : ∀ {Γ B x y}
  → x ≢ y
  → ¬ ∃[ A ] ( Γ ϑ x ⦂ A )
    ----------------------------
  → ¬ ∃[ A ] ( Γ , y ⦂ B ϑ x ⦂ A )
extϑ x≢y _ ⟨ A , Z ⟩     = x≢y refl
extϑ _ ¬∃ ⟨ A , S _ ∋x ⟩ = ¬∃ ⟨ A , ∋x ⟩
```

Given a type `A` and evidence that `Γ , y ⦂ B ϑ x ⦂ A` holds, we must demonstrate a contradiction. If the judgment holds by `Z`, then we must have that `x` and `y` are the same, which contradicts the first assumption. If the judgment holds by `S _ ⊢x` then `⊢x` provides evidence that `Γ ϑ x ⦂ A`, which contradicts the second assumption.

Given a context `Γ` and a variable `x`, we decide whether there exists a type `A` such that `Γ ϑ x ⦂ A` holds, or its negation:

```
lookup : ∀ (Γ : Context) (x : Id)
         ------------------------
       → Dec (∃[ A ] ( Γ ϑ x ⦂ A ))
lookup ∅ x           = no (λ ())
lookup (Γ , y ⦂ B) x with x ≟ y
... | yes refl       = yes ⟨ B , Z ⟩
... | no x≢y with lookup Γ x
... | no ¬∃          = no (extϑ x≢y ¬∃)
... | yes ⟨ A , ∋x ⟩ = yes ⟨ A , S x≢y ∋x ⟩
```

Consider the context:

- If it is empty, then trivially there is no possible derivation.
- If it is non-empty, compare the given variable to the most recent binding:
  - If they are identical, we have succeeded, with `Z` as the appropriate derivation.
  - If they differ, we recurse:
    * If lookup fails, we apply `extϑ` to convert the proof there is no derivation from the contained context to the extended context.
    * If lookup succeeds, we extend the derivation with `S`.

Promoting negations

For each possible term form, we need to show that if one of its components fails to type, then the whole fails to type. Most of these results are easy to demonstrate inline, but we provide auxiliary functions for a couple of the trickier cases.

If `Γ ⊢ L ↑ A ⇒ B` holds but `Γ ⊢ M ↓ A` does not hold, then there is no term `B′` such that `Γ ⊢ L · M ↑ B′` holds:

```
¬arg : ∀ {Γ A B L M}
  → Γ ⊢ L ↑ A ⇒ B
  → ¬ Γ ⊢ M ↓ A
    ----------------------------
  → ¬ ∃[ B′ ] ( Γ ⊢ L · M ↑ B′ )
¬arg ⊢L ¬⊢M ⟨ B′ , ⊢L′ · ⊢M′ ⟩ rewrite dom≡ (uniq-↑ ⊢L ⊢L′) = ¬⊢M ⊢M′
```

Let `⊢L` be evidence that `Γ ⊢ L ↑ A ⇒ B` holds and `¬⊢M` be evidence that `Γ ⊢ M ↓ A` does not hold. Given a type `B′` and evidence that `Γ ⊢ L · M ↑ B′` holds, we must demonstrate a contradiction. The evidence must take the form `⊢L′ · ⊢M′`, where `⊢L′` is evidence that `Γ ⊢ L ↑ A′ ⇒ B′` and `⊢M′` is evidence that `Γ ⊢ M ↓ A′`. By `uniq-↑` applied to `⊢L` and `⊢L′`, we know that `A ⇒ B ≡ A′ ⇒ B′`, and hence that `A ≡ A′`, which means that `¬⊢M` and `⊢M′` yield a contradiction. Without the `rewrite` clause, Agda would not allow us to derive a contradiction between `¬⊢M` and `⊢M′`, since one concerns type `A` and the other type `A′`.

If `Γ ⊢ M ↑ A` holds and `A ≠ B`, then `Γ ⊢ (M ↑) ↓ B` does not hold:

```
¬switch : ∀ {Γ M A B}
  → Γ ⊢ M ↑ A
  → A ≠ B
    ---------------
  → ¬ Γ ⊢ (M ↑) ↓ B
¬switch ⊢M A≠B (⊢↑ ⊢M′ A′≡B) rewrite uniq-↑ ⊢M ⊢M′ = A≠B A′≡B
```

Let `⊢M` be evidence that `Γ ⊢ M ↑ A` holds, and `A≠B` be evidence that `A ≠ B`. Given evidence that `Γ ⊢ (M ↑) ↓ B` holds, we must demonstrate a contradiction. The evidence must take the form `⊢↑ ⊢M′ A′≡B`, where `⊢M′` is evidence that `Γ ⊢ M ↑ A′` and `A′≡B` is evidence that `A′≡B`. By `uniq-↑` applied to `⊢M` and `⊢M′` we know that `A ≡ A′`, which means that `A≠B` and `A′≡B` yield a contradiction. Without the `rewrite` clause, Agda would not allow us to derive a contradiction between `A≠B` and `A′≡B`, since one concerns type `A` and the other type `A′`.

Synthesize and inherit types

The table has been set and we are ready for the main course. We define two mutually recursive functions, one for synthesis and one for inheritance. Synthesis is given a context `Γ` and a synthesis term `M` and either returns a type `A` and evidence that `Γ ⊢ M ↑ A`, or its negation. Inheritance is given a context `Γ`, an inheritance term `M`, and a type `A` and either returns evidence that `Γ ⊢ M ↓ A`, or its negation:

```
synthesize : ∀ (Γ : Context) (M : Term⁺)
             --------------------------
           → Dec (∃[ A ]( Γ ⊢ M ↑ A ))

inherit : ∀ (Γ : Context) (M : Term⁻) (A : Type)
  ---------------
  → Dec (Γ ⊢ M ↓ A)
```

We first consider the code for synthesis:

```
synthesize Γ (` x) with lookup Γ x
... | no ¬∃          = no (λ{ ⟨ A , ⊢` ∋x ⟩ → ¬∃ ⟨ A , ∋x ⟩ })
... | yes ⟨ A , ∋x ⟩  = yes ⟨ A , ⊢` ∋x ⟩
synthesize Γ (L · M) with synthesize Γ L
... | no ¬∃          = no (λ{ ⟨ _ , ⊢L · _  ⟩ → ¬∃ ⟨ _ , ⊢L ⟩ })
... | yes ⟨ `ℕ , ⊢L ⟩ = no (λ{ ⟨ _ , ⊢L′ · _ ⟩ → ℕ≠⇒ (uniq-↑ ⊢L ⊢L′) })
... | yes ⟨ A ⇒ B , ⊢L ⟩ with inherit Γ M A
... | no ¬⊢M         = no (¬arg ⊢L ¬⊢M)
... | yes ⊢M         = yes ⟨ B , ⊢L · ⊢M ⟩
synthesize Γ (M ↓ A) with inherit Γ M A
```

```
... | no ¬⊢M          = no (λ{ ⟨ _ , ⊢↓ ⊢M ⟩ → ¬⊢M ⊢M })
... | yes ⊢M         = yes ⟨ A , ⊢↓ ⊢M ⟩
```

There are three cases:

- If the term is a variable `` ` x ``, we use lookup as defined above:
  - If it fails, then `¬∃` is evidence that there is no `A` such that `Γ ∋ x ⦂ A` holds. Evidence that `Γ ⊢ ` x ↑ A` holds must have the form `⊢` ∋x`, where `∋x` is evidence that `Γ ∋ x ⦂ A`, which yields a contradiction.
  - If it succeeds, then `∋x` is evidence that `Γ ∋ x ⦂ A`, and hence `⊢′ ∋x` is evidence that `Γ ⊢ ` x ↑ A`.
- If the term is an application `L · M`, we recurse on the function `L`:
  - If it fails, then `¬∃` is evidence that there is no type such that `Γ ⊢ L ↑ _` holds. Evidence that `Γ ⊢ L · M ↑ _` holds must have the form `⊢L · _`, where `⊢L` is evidence that `Γ ⊢ L ↑ _`, which yields a contradiction.
  - If it succeeds, there are two possibilities:
    * One is that `⊢L` is evidence that `Γ ⊢ L ⦂ `ℕ`. Evidence that `Γ ⊢ L · M ↑ _` holds must have the form `⊢L′ · _` where `⊢L′` is evidence that `Γ ⊢ L ↑ A ⇒ B` for some types `A` and `B`. Applying `uniq-↑` to `⊢L` and `⊢L′` yields a contradiction, since `` `ℕ `` cannot equal `A ⇒ B`.
    * The other is that `⊢L` is evidence that `Γ ⊢ L ↑ A ⇒ B`, in which case we recurse on the argument `M`:
      · If it fails, then `¬⊢M` is evidence that `Γ ⊢ M ↓ A` does not hold. By `¬arg` applied to `⊢L` and `¬⊢M`, it follows that `Γ ⊢ L · M ↑ B` cannot hold.
      · If it succeeds, then `⊢M` is evidence that `Γ ⊢ M ↓ A`, and `⊢L · ⊢M` provides evidence that `Γ ⊢ L · M ↑ B`.
- If the term is a switch `M ↓ A` from synthesised to inherited, we recurse on the subterm `M`, supplying type `A` by inheritance:
  - If it fails, then `¬⊢M` is evidence that `Γ ⊢ M ↓ A` does not hold. Evidence that `Γ ⊢ (M ↓ A) ↑ A` holds must have the form `⊢↓ ⊢M` where `⊢M` is evidence that `Γ ⊢ M ↓ A` holds, which yields a contradiction.
  - If it succeeds, then `⊢M` is evidence that `Γ ⊢ M ↓ A`, and `⊢↓ ⊢M` provides evidence that `Γ ⊢ (M ↓ A) ↑ A`.

We next consider the code for inheritance:

```
inherit Γ (ƛ x ⇒ N) `ℕ      = no (λ())
inherit Γ (ƛ x ⇒ N) (A ⇒ B) with inherit (Γ , x ⦂ A) N B
... | no ¬⊢N                = no (λ{ (⊢ƛ ⊢N) → ¬⊢N ⊢N })
... | yes ⊢N                = yes (⊢ƛ ⊢N)
inherit Γ `zero `ℕ          = yes ⊢zero
inherit Γ `zero (A ⇒ B)     = no (λ())
inherit Γ (`suc M) `ℕ with inherit Γ M `ℕ
... | no ¬⊢M                = no (λ{ (⊢suc ⊢M) → ¬⊢M ⊢M })
... | yes ⊢M                = yes (⊢suc ⊢M)
```

```
inherit Γ (`suc M) (A ⇒ B) = no (λ())
inherit Γ (`case L [zero⇒ M |suc x ⇒ N ]) A with synthesize Γ L
... | no ¬∃                 = no (λ{ (⊢case ⊢L _ _) → ¬∃ ⟨ `ℕ , ⊢L ⟩})
... | yes ⟨ _ ⇒ _ , ⊢L ⟩    = no (λ{ (⊢case ⊢L′ _ _) → ℕ≠⇒ (uniq-↑ ⊢L′ ⊢L) })
... | yes ⟨ `ℕ , ⊢L ⟩ with inherit Γ M A
... | no ¬⊢M                = no (λ{ (⊢case _ ⊢M _) → ¬⊢M ⊢M })
... | yes ⊢M with inherit (Γ , x ∶ `ℕ) N A
...     | no ¬⊢N            = no (λ{ (⊢case _ _ ⊢N) → ¬⊢N ⊢N })
...     | yes ⊢N            = yes (⊢case ⊢L ⊢M ⊢N)
inherit Γ (μ x ⇒ N) A with inherit (Γ , x ∶ A) N A
... | no ¬⊢N                = no (λ{ (⊢μ ⊢N) → ¬⊢N ⊢N })
... | yes ⊢N                = yes (⊢μ ⊢N)
inherit Γ (M ↑) B with synthesize Γ M
... | no ¬∃                 = no (λ{ (⊢↑ ⊢M _) → ¬∃ ⟨ _ , ⊢M ⟩ })
... | yes ⟨ A , ⊢M ⟩ with A ≟Tp B
... | no A≠B                = no (¬switch ⊢M A≠B)
... | yes A≡B               = yes (⊢↑ ⊢M A≡B)
```

We consider only the cases for abstraction and and for switching from inherited to synthesized:

- If the term is an abstraction `ƛ x ⇒ N` and the inherited type is `` `ℕ ``, then it is trivial that `Γ ⊢ (ƛ x ⇒ N) ↓ `ℕ` cannot hold.
- If the term is an abstraction `ƛ x ⇒ N` and the inherited type is `A ⇒ B`, then we recurse with context `Γ , x ∶ A` on subterm `N` inheriting type `B`:
  - If it fails, then `¬⊢N` is evidence that `Γ , x ∶ A ⊢ N ↓ B` does not hold. Evidence that `Γ ⊢ (ƛ x ⇒ N) ↓ A ⇒ B` holds must have the form `⊢ƛ ⊢N` where `⊢N` is evidence that `Γ , x ∶ A ⊢ N ↓ B`, which yields a contradiction.
  - If it succeeds, then `⊢N` is evidence that `Γ , x ∶ A ⊢ N ↓ B` holds, and `⊢ƛ ⊢N` provides evidence that `Γ ⊢ (ƛ x ⇒ N) ↓ A ⇒ B`.
- If the term is a switch `M ↑` from inherited to synthesised, we recurse on the subterm `M`:
  - If it fails, then `¬∃` is evidence there is no `A` such that `Γ ⊢ M ↑ A` holds. Evidence that `Γ ⊢ (M ↑) ↓ B` holds must have the form `⊢↑ ⊢M _` where `⊢M` is evidence that `Γ ⊢ M ↑ _`, which yields a contradiction.
  - If it succeeds, then `⊢M` is evidence that `Γ ⊢ M ↑ A` holds. We apply `_≟Tp_` do decide whether `A` and `B` are equal:
    - If it fails, then `A≠B` is evidence that `A ≠ B`. By `¬switch` applied to `⊢M` and `A≠B` it follow that `Γ ⊢ (M ↑) ↓ B` cannot hold.
    - If it succeeds, then `A≡B` is evidence that `A ≡ B`, and `⊢↑ ⊢M A≡B` provides evidence that `Γ ⊢ (M ↑) ↓ B`.

The remaining cases are similar, and their code can pretty much be read directly from the corresponding typing rules.

Testing the example terms

First, we copy a function introduced earlier that makes it easy to compute the evidence that two variable names are distinct:

```
_≠_ : ∀ (x y : Id) → x ≢ y
x ≠ y with x ≟ y
... | no x≢y = x≢y
... | yes _  = ⊥-elim impossible
  where postulate impossible : ⊥
```

Here is the result of typing two plus two on naturals:

```
⊢2+2 : ∅ ⊢ 2+2 ↑ `ℕ
⊢2+2 =
  (⊢↓
    (⊢μ
      (⊢ƛ
        (⊢ƛ
          (⊢case (⊢` (S ("m" ≠ "n") Z)) (⊢↑ (⊢` Z) refl)
            (⊢suc
              (⊢↑
                (⊢`
                  (S ("p" ≠ "m")
                    (S ("p" ≠ "n")
                      (S ("p" ≠ "m") Z)))
                · ⊢↑ (⊢` Z) refl
                · ⊢↑ (⊢` (S ("n" ≠ "m") Z)) refl)
               refl))))))
    · ⊢suc (⊢suc ⊢zero)
    · ⊢suc (⊢suc ⊢zero))
```

We confirm that synthesis on the relevant term returns natural as the type and the above derivation:

```
_ : synthesize ∅ 2+2 ≡ yes ⟨ `ℕ , ⊢2+2 ⟩
_ = refl
```

Indeed, the above derivation was computed by evaluating the term on the left, with minor editing of the result. The only editing required was to replace Agda's representation of the evidence that two strings are unequal (which it cannot print nor read) by equivalent calls to `_≠_`.

Here is the result of typing two plus two with Church numerals:

```
⊢2+2ᶜ : ∅ ⊢ 2+2ᶜ ↑ `ℕ
⊢2+2ᶜ =
  ⊢↓
  (⊢ƛ
    (⊢ƛ
      (⊢ƛ
        (⊢ƛ
          (⊢↑
            (⊢`
              (S ("m" ≠ "z")
                (S ("m" ≠ "s")
                  (S ("m" ≠ "n") Z)))
              · ⊢↑ (⊢` (S ("s" ≠ "z") Z)) refl
              ·
              ⊢↑
              (⊢`
                (S ("n" ≠ "z")
                  (S ("n" ≠ "s") Z))
                · ⊢↑ (⊢` (S ("s" ≠ "z") Z)) refl
```

```
            · ⊢↑ (⊢` Z) refl)
          refl)
        refl)))))
·
⊢ƛ
(⊢ƛ
  (⊢↑
    (⊢` (S ("s" ≠ "z") Z) ·
     ⊢↑ (⊢` (S ("s" ≠ "z") Z) · ⊢↑ (⊢` Z) refl)
     refl)
    refl))
·
⊢ƛ
(⊢ƛ
  (⊢↑
    (⊢` (S ("s" ≠ "z") Z) ·
     ⊢↑ (⊢` (S ("s" ≠ "z") Z) · ⊢↑ (⊢` Z) refl)
     refl)
    refl))
· ⊢ƛ (⊢suc (⊢↑ (⊢` Z) refl))
· ⊢zero
```

We confirm that synthesis on the relevant term returns natural as the type and the above derivation:

```
_ : synthesize ∅ 2+2ᶜ ≡ yes ⟨ `ℕ , ⊢2+2ᶜ ⟩
_ = refl
```

Again, the above derivation was computed by evaluating the term on the left and editing.

Testing the error cases

It is important not just to check that code works as intended, but also that it fails as intended. Here are checks for several possible errors:

Unbound variable:

```
_ : synthesize ∅ ((ƛ "x" ⇒ ` "y" ↑) ↓ (`ℕ ⇒ `ℕ)) ≡ no _
_ = refl
```

Argument in application is ill typed:

```
_ : synthesize ∅ (plus · sucᶜ) ≡ no _
_ = refl
```

Function in application is ill typed:

```
_ : synthesize ∅ (plus · sucᶜ · two) ≡ no _
_ = refl
```

Function in application has type natural:

```
_ : synthesize ∅ ((two ↓ `ℕ) · two) ≡ no _
_ = refl
```

Abstraction inherits type natural:

```
_ : synthesize ∅ (two↑ ↓ `ℕ) ≡ no _
_ = refl
```

Zero inherits a function type:

```
_ : synthesize ∅ (`zero ↓ `ℕ ⇒ `ℕ) ≡ no _
_ = refl
```

Successor inherits a function type:

```
_ : synthesize ∅ (two ↓ `ℕ ⇒ `ℕ) ≡ no _
_ = refl
```

Successor of an ill-typed term:

```
_ : synthesize ∅ (`suc two↑ ↓ `ℕ) ≡ no _
_ = refl
```

Case of a term with a function type:

```
_ : synthesize ∅
      ((`case (two↑ ↓ Ch) [zero⇒ `zero |suc "x" ⇒ ` "x" ↑ ] ↓ `ℕ) ) ≡ no _
_ = refl
```

Case of an ill-typed term:

```
_ : synthesize ∅
      ((`case (two↑ ↓ `ℕ) [zero⇒ `zero |suc "x" ⇒ ` "x" ↑ ] ↓ `ℕ) ) ≡ no _
_ = refl
```

Inherited and synthesised types disagree in a switch:

```
_ : synthesize ∅ (((ƛ "x" ⇒ ` "x" ↑) ↓ `ℕ ⇒ (`ℕ ⇒ `ℕ))) ≡ no _
_ = refl
```

Erasure

From the evidence that a decorated term has the correct type it is easy to extract the corresponding intrinsically-typed term. We use the name `DB` to refer to the code in Chapter DeBruijn. It is easy to define an *erasure* function that takes an extrinsic type judgment into the corresponding intrinsically-typed term.

First, we give code to erase a type:

```
||_||Tp : Type → DB.Type
|| `ℕ ||Tp    = DB.`ℕ
|| A ⇒ B ||Tp = || A ||Tp DB.⇒ || B ||Tp
```

It simply renames to the corresponding constructors in module `DB`.

Next, we give the code to erase a context:

```
||_||Cx : Context → DB.Context
|| ∅ ||Cx        = DB.∅
|| Γ , x ⦂ A ||Cx = || Γ ||Cx DB., || A ||Tp
```

It simply drops the variable names.

Next, we give the code to erase a lookup judgment:

```
||_||∋ : ∀ {Γ x A} → Γ ∋ x ⦂ A → || Γ ||Cx DB.∋ || A ||Tp
|| Z ||∋        = DB.Z
|| S x≠ ∋x ||∋ = DB.S || ∋x ||∋
```

It simply drops the evidence that variable names are distinct.

Finally, we give the code to erase a typing judgment. Just as there are two mutually recursive typing judgments, there are two mutually recursive erasure functions:

```
||_||⁺ : ∀ {Γ M A} → Γ ⊢ M ↑ A → || Γ ||Cx DB.⊢ || A ||Tp
||_||⁻ : ∀ {Γ M A} → Γ ⊢ M ↓ A → || Γ ||Cx DB.⊢ || A ||Tp

|| ⊢` ⊢x ||⁺           = DB.` || ⊢x ||∋
|| ⊢L · ⊢M ||⁺         = || ⊢L ||⁺ DB.· || ⊢M ||⁻
|| ⊢↓ ⊢M ||⁺           = || ⊢M ||⁻

|| ⊢ƛ ⊢N ||⁻           = DB.ƛ || ⊢N ||⁻
|| ⊢zero ||⁻          = DB.`zero
|| ⊢suc ⊢M ||⁻        = DB.`suc || ⊢M ||⁻
|| ⊢case ⊢L ⊢M ⊢N ||⁻ = DB.case || ⊢L ||⁺ || ⊢M ||⁻ || ⊢N ||⁻
|| ⊢μ ⊢M ||⁻          = DB.μ || ⊢M ||⁻
|| ⊢↑ ⊢M refl ||⁻     = || ⊢M ||⁺
```

Erasure replaces constructors for each typing judgment by the corresponding term constructor from `DB`. The constructors that correspond to switching from synthesized to inherited or vice versa are dropped.

We confirm that the erasure of the type derivations in this chapter yield the corresponding intrinsically-typed terms from the earlier chapter:

```
_ : || ⊢2+2 ||⁺ ≡ DB.2+2
_ = refl

_ : || ⊢2+2ᶜ ||⁺ ≡ DB.2+2ᶜ
_ = refl
```

Thus, we have confirmed that bidirectional type inference converts decorated versions of the lambda terms from Chapter Lambda to the intrinsically-typed terms of Chapter DeBruijn.

Exercise `inference-multiplication` (recommended)

Apply inference to your decorated definition of multiplication from exercise `bidirectional-mul`, and show that erasure of the inferred typing yields your definition of multiplication from Chapter DeBruijn.

```
-- Your code goes here
```

Exercise `inference-products` (recommended)

Using your rules from exercise `bidirectional-products`, extend bidirectional inference to include products.

```
-- Your code goes here
```

Exercise `inference-rest` (stretch)

Extend the bidirectional type rules to include the rest of the constructs from Chapter More.

```
-- Your code goes here
```

Bidirectional inference in Agda

Agda itself uses bidirectional inference. This explains why constructors can be overloaded while other defined names cannot — here by *overloaded* we mean that the same name can be used for constructors of different types. Constructors are typed by inheritance, and so the name is available when resolving the constructor, whereas variables are typed by synthesis, and so each variable must have a unique type.

Most top-level definitions in Agda are of functions, which are typed by inheritance, which is why Agda requires a type declaration for those definitions. A definition with a right-hand side that is a term typed by synthesis, such as an application, does not require a type declaration.

```
answer = 6 * 7
```

Unicode

This chapter uses the following unicode:

```
↓  U+2193:  DOWNWARDS ARROW (\d)
↑  U+2191:  UPWARDS ARROW (\u)
∥  U+2225:  PARALLEL TO (\||)
```

Chapter 17

Untyped: Untyped lambda calculus with full normalisation

```
module plfa.part2.Untyped where
```

In this chapter we play with variations on a theme:

- Previous chapters consider intrinsically-typed calculi; here we consider one that is untyped but intrinsically scoped.
- Previous chapters consider call-by-value calculi; here we consider call-by-name.
- Previous chapters consider *weak head normal form*, where reduction stops at a lambda abstraction; here we consider *full normalisation*, where reduction continues underneath a lambda.
- Previous chapters consider *deterministic* reduction, where there is at most one redex in a given term; here we consider *non-deterministic* reduction where a term may contain many redexes and any one of them may reduce.
- Previous chapters consider reduction of *closed* terms, those with no free variables; here we consider *open* terms, those which may have free variables.
- Previous chapters consider lambda calculus extended with natural numbers and fixpoints; here we consider a tiny calculus with just variables, abstraction, and application, in which the other constructs may be encoded.

In general, one may mix and match these features, save that full normalisation requires open terms and encoding naturals and fixpoints requires being untyped. The aim of this chapter is to give some appreciation for the range of different lambda calculi one may encounter.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; trans; cong)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _+_; _∸_)
open import Data.Product using (_×_) renaming (_,_ to ⟨_,_⟩)
open import Data.Unit using (⊤; tt)
open import Function using (_∘_)
```

```
open import Function.Equivalence using (_⇔_; equivalence)
open import Relation.Nullary using (¬_; Dec; yes; no)
open import Relation.Nullary.Decidable using (map)
open import Relation.Nullary.Negation using (contraposition)
open import Relation.Nullary.Product using (_×-dec_)
```

Untyped is Uni-typed

Our development will be close to that in Chapter DeBruijn, save that every term will have exactly the same type, written `★` and pronounced "any". This matches a slogan introduced by Dana Scott and echoed by Robert Harper: "Untyped is Uni-typed". One consequence of this approach is that constructs which previously had to be given separately (such as natural numbers and fixpoints) can now be defined in the language itself.

Syntax

First, we get all our infix declarations out of the way:

```
infix 4  _⊢_
infix 4  _∋_
infixl 5 _,_

infix 6  ↓_
infix 6  ′_
infixl 7 _·_
```

Types

We have just one type:

```
data Type : Set where
  ★ : Type
```

Exercise (`Type≃⊤`) (practice)

Show that `Type` is isomorphic to `⊤`, the unit type.

```
-- Your code goes here
```

Contexts

As before, a context is a list of types, with the type of the most recently bound variable on the right:

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

We let `Γ` and `Δ` range over contexts.

Exercise (`Context≃ℕ`) (practice)

Show that `Context` is isomorphic to `ℕ`.

```
-- Your code goes here
```

Variables and the lookup judgment

Intrinsically-scoped variables correspond to the lookup judgment. The rules are as before:

```
data _∋_ : Context → Type → Set where

  Z : ∀ {Γ A}
    ---------
  → Γ , A ∋ A

  S_ : ∀ {Γ A B}
  → Γ ∋ A
    ---------
  → Γ , B ∋ A
```

We could write the rules with all instances of `A` and `B` replaced by `★`, but arguably it is clearer not to do so.

Because `★` is the only type, the judgment doesn't guarantee anything useful about types. But it does ensure that all variables are in scope. For instance, we cannot use `S S Z` in a context that only binds two variables.

Terms and the scoping judgment

Intrinsically-scoped terms correspond to the typing judgment, but with `★` as the only type. The result is that we check that terms are well scoped — that is, that all variables they mention are in scope — but not that they are well typed:

```
data _⊢_ : Context → Type → Set where
  `_ : ∀ {Γ A}
  → Γ ∋ A
    -----
  → Γ ⊢ A
  ƛ_ : ∀ {Γ}
  → Γ , ★ ⊢ ★
    ---------
  → Γ ⊢ ★
```

```
_·_ : ∀ {Γ}
  → Γ ⊢ ★
  → Γ ⊢ ★
    ------
  → Γ ⊢ ★
```

Now we have a tiny calculus, with only variables, abstraction, and application. Below we will see how to encode naturals and fixpoints into this calculus.

Writing variables as numerals

As before, we can convert a natural to the corresponding de Bruijn index. We no longer need to lookup the type in the context, since every variable has the same type:

```
count : ∀ {Γ} → ℕ → Γ ∋ ★
count {Γ , ★} zero    = Z
count {Γ , ★} (suc n) = S (count n)
count {∅} _           = ⊥-elim impossible
  where postulate impossible : ⊥
```

We can then introduce a convenient abbreviation for variables:

```
#_ : ∀ {Γ} → ℕ → Γ ⊢ ★
# n = ` count n
```

Test examples

Our only example is computing two plus two on Church numerals:

```
twoᶜ : ∀ {Γ} → Γ ⊢ ★
twoᶜ = ƛ ƛ (# 1 · (# 1 · # 0))

fourᶜ : ∀ {Γ} → Γ ⊢ ★
fourᶜ = ƛ ƛ (# 1 · (# 1 · (# 1 · (# 1 · # 0))))

plusᶜ : ∀ {Γ} → Γ ⊢ ★
plusᶜ = ƛ ƛ ƛ ƛ (# 3 · # 1 · (# 2 · # 1 · # 0))

2+2ᶜ : ∅ ⊢ ★
2+2ᶜ = plusᶜ · twoᶜ · twoᶜ
```

Before, reduction stopped when we reached a lambda term, so we had to compute `plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero` to ensure we reduced to a representation of the natural four. Now, reduction continues under lambda, so we don't need the extra arguments. It is convenient to define a term to represent four as a Church numeral, as well as two.

Renaming

Our definition of renaming is as before. First, we need an extension lemma:

```
ext : ∀ {Γ Δ} → (∀ {A} → Γ ∋ A → Δ ∋ A)
    ----------------------------------
  → (∀ {A B} → Γ , B ∋ A → Δ , B ∋ A)
ext ρ Z     = Z
ext ρ (S x) = S (ρ x)
```

We could replace all instances of `A` and `B` by `★`, but arguably it is clearer not to do so.

Now it is straightforward to define renaming:

```
rename : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ∋ A)
    ------------------------
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
rename ρ (` x)   = ` (ρ x)
rename ρ (ƛ N)   = ƛ (rename (ext ρ) N)
rename ρ (L · M) = (rename ρ L) · (rename ρ M)
```

This is exactly as before, save that there are fewer term forms.

Simultaneous substitution

Our definition of substitution is also exactly as before. First we need an extension lemma:

```
exts : ∀ {Γ Δ} → (∀ {A} → Γ ∋ A → Δ ⊢ A)
  ---------------------------------
  → (∀ {A B} → Γ , B ∋ A → Δ , B ⊢ A)
exts σ Z     = ` Z
exts σ (S x) = rename S_ (σ x)
```

Again, we could replace all instances of `A` and `B` by `★`.

Now it is straightforward to define substitution:

```
subst : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ⊢ A)
    ------------------------
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
subst σ (` k)   = σ k
subst σ (ƛ N)   = ƛ (subst (exts σ) N)
subst σ (L · M) = (subst σ L) · (subst σ M)
```

Again, this is exactly as before, save that there are fewer term forms.

Single substitution

It is easy to define the special case of substitution for one free variable:

```
subst-zero : ∀ {Γ B} → (Γ ⊢ B) → ∀ {A} → (Γ , B ∋ A) → (Γ ⊢ A)
subst-zero M Z     = M
subst-zero M (S x) = ` x
```

```
_[_] : ∀ {Γ A B}
         → Γ , B ⊢ A
         → Γ ⊢ B
           ---------
         → Γ ⊢ A
_[_] {Γ} {A} {B} N M = subst {Γ , B} {Γ} (subst-zero M) {A} N
```

Neutral and normal terms

Reduction continues until a term is fully normalised. Hence, instead of values, we are now interested in *normal forms*. Terms in normal form are defined by mutual recursion with *neutral* terms:

```
data Neutral : ∀ {Γ A} → Γ ⊢ A → Set
data Normal : ∀ {Γ A} → Γ ⊢ A → Set
```

Neutral terms arise because we now consider reduction of open terms, which may contain free variables. A term is neutral if it is a variable or a neutral term applied to a normal term:

```
data Neutral where

  `_ : ∀ {Γ A} (x : Γ ∋ A)
     -------------
   → Neutral (` x)

  _·_ : ∀ {Γ} {L M : Γ ⊢ ★}
    → Neutral L
    → Normal M
      ---------------
    → Neutral (L · M)
```

A term is a normal form if it is neutral or an abstraction where the body is a normal form. We use `´_` to label neutral terms. Like `` `_ ``, it is unobtrusive:

```
data Normal where

  ´_ : ∀ {Γ A} {M : Γ ⊢ A}
    → Neutral M
      ---------
    → Normal M

  ƛ_ : ∀ {Γ} {N : Γ , ★ ⊢ ★}
    → Normal N
      ------------
    → Normal (ƛ N)
```

We introduce a convenient abbreviation for evidence that a variable is neutral:

```
#´_ : ∀ {Γ} (n : ℕ) → Neutral {Γ} (# n)
#´ n = ` count n
```

For example, here is the evidence that the Church numeral two is in normal form:

```
_ : Normal (twoᶜ {∅})
_ = ƛ ƛ (` #′ 1 · (` #′ 1 · (` #′ 0)))
```

The evidence that a term is in normal form is almost identical to the term itself, decorated with some additional primes to indicate neutral terms, and using `#′` in place of `#`

Reduction step

The reduction rules are altered to switch from call-by-value to call-by-name and to enable full normalisation:

- The rule `ξ₁` remains the same as it was for the simply-typed lambda calculus.
- In rule `ξ₂`, the requirement that the term `L` is a value is dropped. So this rule can overlap with `ξ₁` and reduction is *non-deterministic*. One can choose to reduce a term inside either `L` or `M`.
- In rule `β`, the requirement that the argument is a value is dropped, corresponding to call-by-name evaluation. This introduces further non-determinism, as `β` overlaps with `ξ₂` when there are redexes in the argument.
- A new rule `ζ` is added, to enable reduction underneath a lambda.

Here are the formalised rules:

```
infix 2 _⟶_

data _⟶_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  ξ₁ : ∀ {Γ} {L L′ M : Γ ⊢ ★}
    → L ⟶ L′
      ----------------
    → L · M ⟶ L′ · M

  ξ₂ : ∀ {Γ} {L M M′ : Γ ⊢ ★}
    → M ⟶ M′
      ----------------
    → L · M ⟶ L · M′

  β : ∀ {Γ} {N : Γ , ★ ⊢ ★} {M : Γ ⊢ ★}
      ---------------------------------
    → (ƛ N) · M ⟶ N [ M ]

  ζ : ∀ {Γ} {N N′ : Γ , ★ ⊢ ★}
    → N ⟶ N′
      -----------
    → ƛ N ⟶ ƛ N′
```

Exercise (`variant-1`) (practice)

How would the rules change if we want call-by-value where terms normalise completely? Assume that `β` should not permit reduction unless both terms are in normal form.

```
-- Your code goes here
```

Exercise (`variant-2`) (practice)

How would the rules change if we want call-by-value where terms do not reduce underneath lambda? Assume that `β` permits reduction when both terms are values (that is, lambda abstractions). What would `2+2ᶜ` reduce to in this case?

```
-- Your code goes here
```

Reflexive and transitive closure

We cut-and-paste the previous definition:

```
infix 2 _—»_
infix 1 begin_
infixr 2 _—→⟨_⟩_
infix 3 _∎

data _—»_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  _∎ : ∀ {Γ A} (M : Γ ⊢ A)
     --------
    → M —» M

  _—→⟨_⟩_ : ∀ {Γ A} (L : Γ ⊢ A) {M N : Γ ⊢ A}
    → L —→ M
    → M —» N
      ---------
    → L —» N

begin_ : ∀ {Γ} {A} {M N : Γ ⊢ A}
  → M —» N
    ------
  → M —» N
begin M—»N = M—»N
```

Example reduction sequence

Here is the demonstration that two plus two is four:

```
_ : 2+2ᶜ —» fourᶜ
_ =
  begin
    plusᶜ · twoᶜ · twoᶜ
  —→⟨ ξ₁ β ⟩
    (ƛ ƛ ƛ twoᶜ · # 1 · (# 2 · # 1 · # 0)) · twoᶜ
  —→⟨ β ⟩
    ƛ ƛ twoᶜ · # 1 · (twoᶜ · # 1 · # 0)
```

```
  ⟶⟨ ζ (ζ (ξ₁ β)) ⟩
    ƛ ƛ ((ƛ # 2 · (# 2 · # 0)) · (twoᶜ · # 1 · # 0))
  ⟶⟨ ζ (ζ β) ⟩
    ƛ ƛ # 1 · (# 1 · (twoᶜ · # 1 · # 0))
  ⟶⟨ ζ (ζ (ξ₂ (ξ₂ (ξ₁ β)))) ⟩
    ƛ ƛ # 1 · (# 1 · ((ƛ # 2 · (# 2 · # 0)) · # 0))
  ⟶⟨ ζ (ζ (ξ₂ (ξ₂ β))) ⟩
    ƛ (ƛ # 1 · (# 1 · (# 1 · (# 1 · # 0))))
  ∎
```

After just two steps the top-level term is an abstraction, and `ζ` rules drive the rest of the normalisation.

Progress

Progress adapts. Instead of claiming that every term either is a value or takes a reduction step, we claim that every term is either in normal form or takes a reduction step.

Previously, progress only applied to closed, well-typed terms. We had to rule out terms where we apply something other than a function (such as `` `zero ``) or terms with a free variable. Now we can demonstrate it for open, well-scoped terms. The definition of normal form permits free variables, and we have no terms that are not functions.

A term makes progress if it can take a step or is in normal form:

```
data Progress {Γ A} (M : Γ ⊢ A) : Set where

  step : ∀ {N : Γ ⊢ A}
    → M ⟶ N
      ----------
    → Progress M

  done :
      Normal M
      ----------
    → Progress M
```

If a term is well scoped then it satisfies progress:

```
progress : ∀ {Γ A} → (M : Γ ⊢ A) → Progress M
progress (` x)       = done (´ ` x)
progress (ƛ N) with progress N
... | step N⟶N′      = step (ζ N⟶N′)
... | done NrmN      = done (ƛ NrmN)
progress (` x · M) with progress M
... | step M⟶M′      = step (ξ₂ M⟶M′)
... | done NrmM      = done (´ (` x) · NrmM)
progress ((ƛ N) · M) = step β
progress (L@(_ · _) · M) with progress L
... | step L⟶L′      = step (ξ₁ L⟶L′)
... | done (´ NeuL) with progress M
... | step M⟶M′      = step (ξ₂ M⟶M′)
... | done NrmM      = done (´ NeuL · NrmM)
```

We induct on the evidence that the term is well scoped:

- If the term is a variable, then it is in normal form. (This contrasts with previous proofs, where the variable case was ruled out by the restriction to closed terms.)
- If the term is an abstraction, recursively invoke progress on the body. (This contrast with previous proofs, where an abstraction is immediately a value.):
  - If it steps, then the whole term steps via `ζ`.
  - If it is in normal form, then so is the whole term.
- If the term is an application, consider the function subterm:
  - If it is a variable, recursively invoke progress on the argument:
    - If it steps, then the whole term steps via `ξ₂`;
    - If it is normal, then so is the whole term.
  - If it is an abstraction, then the whole term steps via `β`.
  - If it is an application, recursively apply progress to the function subterm:
    - If it steps, then the whole term steps via `ξ₁`.
    - If it is normal, recursively apply progress to the argument subterm:
      - If it steps, then the whole term steps via `ξ₂`.
      - If it is normal, then so is the whole term.

The final equation for progress uses an *at pattern* of the form `P@Q`, which matches only if both pattern `P` and pattern `Q` match. Character `@` is one of the few that Agda doesn't allow in names, so spaces are not required around it. In this case, the pattern ensures that `L` is an application.

Evaluation

As previously, progress immediately yields an evaluator.

Gas is specified by a natural number:

```
record Gas : Set where
  constructor gas
  field
    amount : ℕ
```

When our evaluator returns a term `N`, it will either give evidence that `N` is normal or indicate that it ran out of gas:

```
data Finished {Γ A} (N : Γ ⊢ A) : Set where

  done :
    Normal N
    ----------
    → Finished N

  out-of-gas :
    ----------
    Finished N
```

Given a term `L` of type `A`, the evaluator will, for some `N`, return a reduction sequence from `L` to `N` and an indication of whether reduction finished:

```
data Steps : ∀ {Γ A} → Γ ⊢ A → Set where

  steps : ∀ {Γ A} {L N : Γ ⊢ A}
```

```
    → L —↠ N
    → Finished N
      ----------
    → Steps L
```

The evaluator takes gas and a term and returns the corresponding steps:

```
eval : ∀ {Γ A}
  → Gas
  → (L : Γ ⊢ A)
    -----------
  → Steps L
eval (gas zero) L     = steps (L ▮) out-of-gas
eval (gas (suc m)) L with progress L
... | done NrmL       = steps (L ▮) (done NrmL)
... | step {M} L—→M with eval (gas m) M
... | steps M—↠N fin = steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

The definition is as before, save that the empty context `∅` generalises to an arbitrary context `Γ`.

Example

We reiterate our previous example. Two plus two is four, with Church numerals:

```
_ : eval (gas 100) 2+2ᶜ ≡
  steps
    ((ƛ
       (ƛ
         (ƛ
           (` (S (S (S Z)))) · (` (S Z)) ·
           ((` (S (S Z))) · (` (S Z)) · (` Z))))))
      · (ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z))))
      · (ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z))))
    —→⟨ ξ₁ β ⟩
      (ƛ
        (ƛ
          (ƛ
            (ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z)))) · (` (S Z)) ·
            ((` (S (S Z))) · (` (S Z)) · (` Z)))))
      · (ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z))))
    —→⟨ β ⟩
      ƛ
      (ƛ
        (ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z)))) · (` (S Z)) ·
        ((ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z)))) · (` (S Z)) · (` Z)))
    —→⟨ ζ (ζ (ξ₁ β)) ⟩
      ƛ
      (ƛ
        (ƛ (` (S (S Z))) · ((` (S (S Z))) · (` Z))) ·
        ((ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z)))) · (` (S Z)) · (` Z)))
    —→⟨ ζ (ζ β) ⟩
      ƛ
      (ƛ
        (` (S Z)) ·
```

```
      ((` (S Z)) ·
        ((ƛ (ƛ (` (S Z)) · ((` (S Z)) · (` Z)))) · (` (S Z)) · (` Z))))
  —→⟨ ζ (ζ (ξ₂ (ξ₂ (ξ₁ β)))) ⟩
    ƛ
    (ƛ
      (` (S Z)) ·
      ((` (S Z)) ·
        ((ƛ (` (S (S Z))) · ((` (S (S Z))) · (` Z))) · (` Z))))
  —→⟨ ζ (ζ (ξ₂ (ξ₂ β))) ⟩
    ƛ (ƛ (` (S Z)) · ((` (S Z)) · ((` (S Z)) · ((` (S Z)) · (` Z)))))
  ∎)
  (done
    (ƛ
      (ƛ
        (′
          (` (S Z)) ·
          (′ (` (S Z)) · (′ (` (S Z)) · (′ (` (S Z)) · (′ (` Z))))))))))
_ = refl
```

Naturals and fixpoint

We could simulate naturals using Church numerals, but computing predecessor is tricky and expensive. Instead, we use a different representation, called Scott numerals, where a number is essentially defined by the expression that corresponds to its own case statement.

Recall that Church numerals apply a given function for the corresponding number of times. Using named terms, we represent the first three Church numerals as follows:

```
zero  =  ƛ s ⇒ ƛ z ⇒ z
one   =  ƛ s ⇒ ƛ z ⇒ s · z
two   =  ƛ s ⇒ ƛ z ⇒ s · (s · z)
```

In contrast, for Scott numerals, we represent the first three naturals as follows:

```
zero = ƛ s ⇒ ƛ z ⇒ z
one  = ƛ s ⇒ ƛ z ⇒ s · zero
two  = ƛ s ⇒ ƛ z ⇒ s · one
```

Each representation expects two arguments, one corresponding to the successor branch of the case (it expects an additional argument, the predecessor of the current argument) and one corresponding to the zero branch of the case. (The cases could be in either order. We put the successor case first to ease comparison with Church numerals.)

Here is the Scott representation of naturals encoded with de Bruijn indexes:

```
`zero : ∀ {Γ} → (Γ ⊢ ★)
`zero = ƛ ƛ (# 0)

`suc_ : ∀ {Γ} → (Γ ⊢ ★) → (Γ ⊢ ★)
`suc_ M = (ƛ ƛ ƛ (# 1 · # 2)) · M

case : ∀ {Γ} → (Γ ⊢ ★) → (Γ ⊢ ★) → (Γ , ★ ⊢ ★) → (Γ ⊢ ★)
case L M N = L · (ƛ N) · M
```

Here we have been careful to retain the exact form of our previous definitions. The successor branch expects an additional variable to be in scope (as indicated by its type), so it is converted to an ordinary term using lambda abstraction.

Applying successor to the zero indeed reduces to the Scott numeral for one.

```
_ : eval (gas 100) (`suc_ {∅} `zero) ≡
  steps
    ((ƛ (ƛ (ƛ # 1 · # 2))) · (ƛ (ƛ # 0))
  ⟶⟨ β ⟩
      ƛ (ƛ # 1 · (ƛ (ƛ # 0)))
  ∎)
  (done (ƛ (ƛ (´ (` (S Z)) · (ƛ (ƛ (´ (` Z))))))))
_ = refl
```

We can also define fixpoint. Using named terms, we define:

```
μ f = (ƛ x ⇒ f · (x · x)) · (ƛ x ⇒ f · (x · x))
```

This works because:

```
  μ f
≡
  (ƛ x ⇒ f · (x · x)) · (ƛ x ⇒ f · (x · x))
⟶
  f · ((ƛ x ⇒ f · (x · x)) · (ƛ x ⇒ f · (x · x)))
≡
  f · (μ f)
```

With de Bruijn indices, we have the following:

```
μ_ : ∀ {Γ} → (Γ , ★ ⊢ ★) → (Γ ⊢ ★)
μ N = (ƛ ((ƛ (# 1 · (# 0 · # 0))) · (ƛ (# 1 · (# 0 · # 0))))) · (ƛ N)
```

The argument to fixpoint is treated similarly to the successor branch of case.

We can now define two plus two exactly as before:

```
infix 5 μ_

two : ∀ {Γ} → Γ ⊢ ★
two = `suc `suc `zero

four : ∀ {Γ} → Γ ⊢ ★
four = `suc `suc `suc `suc `zero

plus : ∀ {Γ} → Γ ⊢ ★
plus = μ ƛ ƛ (case (# 1) (# 0) (`suc (# 3 · # 0 · # 1)))
```

Because `` `suc `` is now a defined term rather than primitive, it is no longer the case that `plus · two · two` reduces to `four`, but they do both reduce to the same normal term.

Exercise `plus-eval` (practice)

Use the evaluator to confirm that `plus · two · two` and `four` normalise to the same term.

```
-- Your code goes here
```

Exercise `multiplication-untyped` **(recommended)**

Use the encodings above to translate your definition of multiplication from previous chapters with the Scott representation and the encoding of the fixpoint operator. Confirm that two times two is four.

```
-- Your code goes here
```

Exercise `encode-more` **(stretch)**

Along the lines above, encode all of the constructs of Chapter More, save for primitive numbers, in the untyped lambda calculus.

```
-- Your code goes here
```

Multi-step reduction is transitive

In our formulation of the reflexive transitive closure of reduction, i.e., the `—↠` relation, there is not an explicit rule for transitivity. Instead the relation mimics the structure of lists by providing a case for an empty reduction sequence and a case for adding one reduction to the front of a reduction sequence. The following is the proof of transitivity, which has the same structure as the append function `_++_` on lists.

```
—↠-trans : ∀{Γ}{A}{L M N : Γ ⊢ A}
         → L —↠ M
         → M —↠ N
         → L —↠ N
—↠-trans (M ∎) mn = mn
—↠-trans (L —→⟨ r ⟩ lm) mn = L —→⟨ r ⟩ (—↠-trans lm mn)
```

The following notation makes it convenient to employ transitivity of `—↠`.

```
infixr 2 _—↠⟨_⟩_

_—↠⟨_⟩_ : ∀ {Γ A} (L : Γ ⊢ A) {M N : Γ ⊢ A}
  → L —↠ M
  → M —↠ N
    ---------
  → L —↠ N
L —↠⟨ L—↠M ⟩ M—↠N = —↠-trans L—↠M M—↠N
```

Multi-step reduction is a congruence

Recall from Chapter Induction that a relation `R` is a *congruence* for a given function `f` if it is preserved by that function, i.e., if `R x y` then `R (f x) (f y)`. The term constructors `ƛ_` and `_·_` are functions, and so the notion of congruence applies to them as well. Furthermore, when a relation is a congruence for all of the term constructors, we say that the relation is a congruence for the language in question, in this case the untyped lambda calculus.

The rules `ξ₁`, `ξ₂`, and `ζ` ensure that the reduction relation is a congruence for the untyped lambda calculus. The multi-step reduction relation `—↠` is also a congruence, which we prove in the following three lemmas.

```
appL-cong : ∀ {Γ} {L L' M : Γ ⊢ ★}
  → L —↠ L'
    ---------------
  → L · M —↠ L' · M
appL-cong {Γ}{L}{L'}{M} (L ∎) = L · M ∎
appL-cong {Γ}{L}{L'}{M} (L —→⟨ r ⟩ rs) = L · M —→⟨ ξ₁ r ⟩ appL-cong rs
```

The proof of `appL-cong` is by induction on the reduction sequence `L —↠ L'`. * Suppose `L —↠ L` by `L ∎`. Then we have `L · M —↠ L · M` by `L · M ∎`. * Suppose `L —↠ L''` by `L —→⟨ r ⟩ rs`, so `L —→ L'` by `r` and `L' —↠ L''` by `rs`. We have `L · M —→ L' · M` by `ξ₁ r` and `L' · M —↠ L'' · M` by the induction hypothesis applied to `rs`. We conclude that `L · M —↠ L'' · M` by putting these two facts together using `_—→⟨_⟩_`.

The proofs of `appR-cong` and `abs-cong` follow the same pattern as the proof for `appL-cong`.

```
appR-cong : ∀ {Γ} {L M M' : Γ ⊢ ★}
  → M —↠ M'
    ---------------
  → L · M —↠ L · M'
appR-cong {Γ}{L}{M}{M'} (M ∎) = L · M ∎
appR-cong {Γ}{L}{M}{M'} (M —→⟨ r ⟩ rs) = L · M —→⟨ ξ₂ r ⟩ appR-cong rs
```

```
abs-cong : ∀ {Γ} {N N' : Γ , ★ ⊢ ★}
         → N —↠ N'
           ----------
         → ƛ N —↠ ƛ N'
abs-cong (M ∎) = ƛ M ∎
abs-cong (L —→⟨ r ⟩ rs) = ƛ L —→⟨ ζ r ⟩ abs-cong rs
```

Unicode

This chapter uses the following unicode:

```
★  U+2605  BLACK STAR (\st)
```

The `\st` command permits navigation among many different stars; the one we use is number 7.

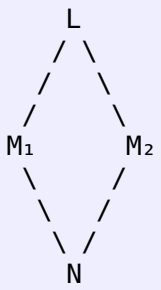
Chapter 18

Confluence: Confluence of untyped lambda calculus

```
module plfa.part2.Confluence where
```

Introduction

In this chapter we prove that beta reduction is *confluent*, a property also known as *Church-Rosser*. That is, if there are reduction sequences from any term `L` to two different terms `M₁` and `M₂`, then there exist reduction sequences from those two terms to some common term `N`. In pictures:



where downward lines are instances of `—↠`.

Confluence is studied in many other kinds of rewrite systems besides the lambda calculus, and it is well known how to prove confluence in rewrite systems that enjoy the *diamond property*, a single-step version of confluence. Let `⇒` be a relation. Then `⇒` has the diamond property if whenever `L ⇒ M₁` and `L ⇒ M₂`, then there exists an `N` such that `M₁ ⇒ N` and `M₂ ⇒ N`. This is just an instance of the same picture above, where downward lines are now instance of `⇒`. If we write `⇒*` for the reflexive and transitive closure of `⇒`, then confluence of `⇒*` follows immediately from the diamond property.

Unfortunately, reduction in the lambda calculus does not satisfy the diamond property. Here is a counter example.

```
(λ x. x x)((λ x. x) a) ⟶ (λ x. x x) a
(λ x. x x)((λ x. x) a) ⟶ ((λ x. x) a) ((λ x. x) a)
```

Both terms can reduce to `a a`, but the second term requires two steps to get there, not one.

To side-step this problem, we'll define an auxiliary reduction relation, called *parallel reduction*, that can perform many reductions simultaneously and thereby satisfy the diamond property. Furthermore, we show that a parallel reduction sequence exists between any two terms if and only if a beta reduction sequence exists between them. Thus, we can reduce the proof of confluence for beta reduction to confluence for parallel reduction.

Imports

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Function using (_∘_)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import plfa.part2.Substitution using (Rename; Subst)
open import plfa.part2.Untyped
  using (_—→_; β; ξ₁; ξ₂; ζ; _—↠_; begin_; _—→⟨_⟩_; _—↠⟨_⟩_; _∎;
  abs-cong; appL-cong; appR-cong; —↠-trans;
  _⊢_; _∋_; `_; #_; _,_; ★; ƛ_; _·_; _[_];
  rename; ext; exts; Z; S_; subst; subst-zero)
```

Parallel Reduction

The parallel reduction relation is defined as follows.

```
infix 2 _⇒_

data _⇒_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  pvar : ∀{Γ A}{x : Γ ∋ A}
    ---------
    → (` x) ⇒ (` x)

  pabs : ∀{Γ}{N N′ : Γ , ★ ⊢ ★}
    → N ⇒ N′
      ----------
    → ƛ N ⇒ ƛ N′

  papp : ∀{Γ}{L L′ M M′ : Γ ⊢ ★}
    → L ⇒ L′
    → M ⇒ M′
      ------------------
    → L · M ⇒ L′ · M′

  pbeta : ∀{Γ}{N N′ : Γ , ★ ⊢ ★}{M M′ : Γ ⊢ ★}
    → N ⇒ N′
    → M ⇒ M′
      ----------------------
    → (ƛ N) · M ⇒ N′ [ M′ ]
```

The first three rules are congruences that reduce each of their parts simultaneously. The last rule reduces a lambda term and term in parallel followed by a beta step.

We remark that the `pabs`, `papp`, and `pbeta` rules perform reduction on all their subexpressions simultaneously. Also, the `pabs` rule is akin to the `ζ` rule and `pbeta` is akin to `β`.

Parallel reduction is reflexive.

```
par-refl : ∀{Γ A}{M : Γ ⊢ A} → M ⇛ M
par-refl {Γ} {A} {` x} = pvar
par-refl {Γ} {⋆} {ƛ N} = pabs par-refl
par-refl {Γ} {⋆} {L · M} = papp par-refl par-refl
```

We define the sequences of parallel reduction as follows.

```
infix 2 _⇛*_
infixr 2 _⇛⟨_⟩_
infix 3 _∎

data _⇛*_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  _∎ : ∀ {Γ A} (M : Γ ⊢ A)
     --------
   → M ⇛* M

  _⇛⟨_⟩_ : ∀ {Γ A} (L : Γ ⊢ A) {M N : Γ ⊢ A}
   → L ⇛ M
   → M ⇛* N
     ---------
   → L ⇛* N
```

Exercise `par-diamond-eg` **(practice)**

Revisit the counter example to the diamond property for reduction by showing that the diamond property holds for parallel reduction in that case.

```
-- Your code goes here
```

Equivalence between parallel reduction and reduction

Here we prove that for any `M` and `N`, `M ⇛* N` if and only if `M ⟶↠ N`. The only-if direction is particularly easy. We start by showing that if `M ⟶ N`, then `M ⇛ N`. The proof is by induction on the reduction `M ⟶ N`.

```
beta-par : ∀{Γ A}{M N : Γ ⊢ A}
  → M ⟶ N
    ------
  → M ⇛ N
beta-par {Γ} {⋆} {L · M} (ξ₁ r) = papp (beta-par {M = L} r) par-refl
beta-par {Γ} {⋆} {L · M} (ξ₂ r) = papp par-refl (beta-par {M = M} r)
beta-par {Γ} {⋆} {(ƛ N) · M} β = pbeta par-refl par-refl
beta-par {Γ} {⋆} {ƛ N} (ζ r) = pabs (beta-par r)
```

With this lemma in hand we complete the only-if direction, that `M ⟶↠ N` implies `M ⇛* N`. The proof is a straightforward induction on the reduction sequence `M ⟶↠ N`.

```
betas-pars : ∀{Γ A} {M N : Γ ⊢ A}
  → M —↠ N
    ------
  → M ⇛* N
betas-pars {Γ} {A} {M₁} {.M₁} (M₁ ∎) = M₁ ∎
betas-pars {Γ} {A} {.L} {N} (L —→⟨ b ⟩ bs) =
  L ⇛⟨ beta-par b ⟩ betas-pars bs
```

Now for the other direction, that `M ⇛* N` implies `M —↠ N`. The proof of this direction is a bit different because it's not the case that `M ⇛ N` implies `M —→ N`. After all, `M ⇛ N` performs many reductions. So instead we shall prove that `M ⇛ N` implies `M —↠ N`.

```
par-betas : ∀{Γ A}{M N : Γ ⊢ A}
  → M ⇛ N
    ------
  → M —↠ N
par-betas {Γ} {A} {.(` _)} (pvar{x = x}) = (` x) ∎
par-betas {Γ} {★} {ƛ N} (pabs p) = abs-cong (par-betas p)
par-betas {Γ} {★} {L · M} (papp {L = L}{L′}{M}{M′} p₁ p₂) =
   begin
   L · M  —↠⟨ appL-cong{M = M} (par-betas p₁) ⟩
   L′ · M —↠⟨ appR-cong (par-betas p₂) ⟩
   L′ · M′
   ∎
par-betas {Γ} {★} {(ƛ N) · M} (pbeta{N′ = N′}{M′ = M′} p₁ p₂) =
   begin
   (ƛ N) · M   —↠⟨ appL-cong{M = M} (abs-cong (par-betas p₁)) ⟩
   (ƛ N′) · M  —↠⟨ appR-cong{L = ƛ N′} (par-betas p₂) ⟩
   (ƛ N′) · M′ —→⟨ β ⟩
     N′ [ M′ ]
   ∎
```

The proof is by induction on `M ⇛ N`.

- Suppose `x ⇛ x`. We immediately have `x —↠ x`.
- Suppose `ƛ N ⇛ ƛ N′` because `N ⇛ N′`. By the induction hypothesis we have `N —↠ N′`. We conclude that `ƛ N —↠ ƛ N′` because `—↠` is a congruence.
- Suppose `L · M ⇛ L′ · M′` because `L ⇛ L′` and `M ⇛ M′`. By the induction hypothesis, we have `L —↠ L′` and `M —↠ M′`. So `L · M —↠ L′ · M` and then `L′ · M —↠ L′ · M′` because `—↠` is a congruence.
- Suppose `(ƛ N) · M ⇛ N′ [ M′ ]` because `N ⇛ N′` and `M ⇛ M′`. By similar reasoning, we have `(ƛ N) · M —↠ (ƛ N′) · M′` which we can following with the β reduction `(ƛ N′) · M′ —→ N′ [ M′ ]`.

With this lemma in hand, we complete the proof that `M ⇛* N` implies `M —↠ N` with a simple induction on `M ⇛* N`.

```
pars-betas : ∀{Γ A} {M N : Γ ⊢ A}
  → M ⇛* N
    ------
  → M —↠ N
pars-betas (M₁ ∎) = M₁ ∎
pars-betas (L ⇛⟨ p ⟩ ps) = —↠-trans (par-betas p) (pars-betas ps)
```

Substitution lemma for parallel reduction

Our next goal is the prove the diamond property for parallel reduction. But to do that, we need to prove that substitution respects parallel reduction. That is, if `N ⇒ N′` and `M ⇒ M′`, then `N [ M ] ⇒ N′ [ M′ ]`. We cannot prove this directly by induction, so we generalize it to: if `N ⇒ N′` and the substitution `σ` pointwise parallel reduces to `τ`, then `subst σ N ⇒ subst τ N′`. We define the notion of pointwise parallel reduction as follows.

```
par-subst : ∀{Γ Δ} → Subst Γ Δ → Subst Γ Δ → Set
par-subst {Γ}{Δ} σ σ′ = ∀{A}{x : Γ ∋ A} → σ x ⇒ σ′ x
```

Because substitution depends on the extension function `exts`, which in turn relies on `rename`, we start with a version of the substitution lemma, called `par-rename`, that is specialized to renamings. The proof of `par-rename` relies on the fact that renaming and substitution commute with one another, which is a lemma that we import from Chapter Substitution and restate here.

```
rename-subst-commute : ∀{Γ Δ}{N : Γ , ⋆ ⊢ ⋆}{M : Γ ⊢ ⋆}{ρ : Rename Γ Δ }
  → (rename (ext ρ) N) [ rename ρ M ] ≡ rename ρ (N [ M ])
rename-subst-commute {N = N} = plfa.part2.Substitution.rename-subst-commute {N = N}
```

Now for the `par-rename` lemma.

```
par-rename : ∀{Γ Δ A} {ρ : Rename Γ Δ} {M M′ : Γ ⊢ A}
  → M ⇒ M′
    ------------------------
  → rename ρ M ⇒ rename ρ M′
par-rename pvar = pvar
par-rename (pabs p) = pabs (par-rename p)
par-rename (papp p₁ p₂) = papp (par-rename p₁) (par-rename p₂)
par-rename {Γ}{Δ}{A}{ρ} (pbeta{Γ}{N}{N′}{M}{M′} p₁ p₂)
   with pbeta (par-rename{ρ = ext ρ} p₁) (par-rename{ρ = ρ} p₂)
... | G rewrite rename-subst-commute{Γ}{Δ}{N′}{M′}{ρ} = G
```

The proof is by induction on `M ⇒ M′`. The first four cases are straightforward so we just consider the last one for `pbeta`.

- Suppose `(ƛ N) · M ⇒ N′ [ M′ ]` because `N ⇒ N′` and `M ⇒ M′`. By the induction hypothesis, we have `rename (ext ρ) N ⇒ rename (ext ρ) N′` and `rename ρ M ⇒ rename ρ M′`. So by `pbeta` we have `(ƛ rename (ext ρ) N) · (rename ρ M) ⇒ (rename (ext ρ) N) [ rename ρ M ]`. However, to conclude we instead need parallel reduction to `rename ρ (N [ M ])`. But thankfully, renaming and substitution commute with one another.

With the `par-rename` lemma in hand, it is straightforward to show that extending substitutions preserves the pointwise parallel reduction relation.

```
par-subst-exts : ∀{Γ Δ} {σ τ : Subst Γ Δ}
  → par-subst σ τ
    ----------------------------------------
  → ∀{B} → par-subst (exts σ {B = B}) (exts τ)
par-subst-exts s {x = Z} = pvar
par-subst-exts s {x = S x} = par-rename s
```

The next lemma that we need for proving that substitution respects parallel reduction is the following which states that simultaneoous substitution commutes with single substitution. We import this lemma from Chapter Substitution and restate it below.

```
subst-commute : ∀{Γ Δ}{N : Γ , ⋆ ⊢ ⋆}{M : Γ ⊢ ⋆}{σ : Subst Γ Δ }
  → subst (exts σ) N [ subst σ M ] ≡ subst σ (N [ M ])
subst-commute {N = N} = plfa.part2.Substitution.subst-commute {N = N}
```

We are ready to prove that substitution respects parallel reduction.

```
subst-par : ∀{Γ Δ A} {σ τ : Subst Γ Δ} {M M′ : Γ ⊢ A}
  → par-subst σ τ → M ⇛ M′
    -------------------------
  → subst σ M ⇛ subst τ M′
subst-par {Γ} {Δ} {A} {σ} {τ} {` x} s pvar = s
subst-par {Γ} {Δ} {A} {σ} {τ} {ƛ N} s (pabs p) =
  pabs (subst-par {σ = exts σ} {τ = exts τ}
         (λ {A}{x} → par-subst-exts s {x = x}) p)
subst-par {Γ} {Δ} {⋆} {σ} {τ} {L · M} s (papp p₁ p₂) =
  papp (subst-par s p₁) (subst-par s p₂)
subst-par {Γ} {Δ} {⋆} {σ} {τ} {(ƛ N) · M} s (pbeta{N′ = N′}{M′ = M′} p₁ p₂)
   with pbeta (subst-par{σ = exts σ}{τ = exts τ}{M = N}
                 (λ{A}{x} → par-subst-exts s {x = x}) p₁)
              (subst-par {σ = σ} s p₂)
... | G rewrite subst-commute{N = N′}{M = M′}{σ = τ} = G
```

We proceed by induction on `M ⇛ M′`.

- Suppose `x ⇛ x`. We conclude that `σ x ⇛ τ x` using the premise `par-subst σ τ`.
- Suppose `ƛ N ⇛ ƛ N′` because `N ⇛ N′`. To use the induction hypothesis, we need `par-subst (exts σ) (exts τ)`, which we obtain by `par-subst-exts`. So we have `subst (exts σ) N ⇛ subst (exts τ) N′` and conclude by rule `pabs`.
- Suppose `L · M ⇛ L′ · M′` because `L ⇛ L′` and `M ⇛ M′`. By the induction hypothesis we have `subst σ L ⇛ subst τ L′` and `subst σ M ⇛ subst τ M′`, so we conclude by rule `papp`.
- Suppose `(ƛ N) · M ⇛ N′ [ M′ ]` because `N ⇛ N′` and `M ⇛ M′`. Again we obtain `par-subst (exts σ) (exts τ)` by `par-subst-exts`. So by the induction hypothesis, we have `subst (exts σ) N ⇛ subst (exts τ) N′` and `subst σ M ⇛ subst τ M′`. Then by rule `pbeta`, we have parallel reduction to `subst (exts τ) N′ [ subst τ M′ ]`. Substitution commutes with itself in the following sense. For any σ, N, and M, we have

  ```
  (subst (exts σ) N) [ subst σ M ] ≡ subst σ (N [ M ])
  ```

  So we have parallel reduction to `subst τ (N′ [ M′ ])`.

Of course, if `M ⇛ M′`, then `subst-zero M` pointwise parallel reduces to `subst-zero M′`.

```
par-subst-zero : ∀{Γ}{A}{M M′ : Γ ⊢ A}
  → M ⇛ M′
  → par-subst (subst-zero M) (subst-zero M′)
par-subst-zero {M} {M′} p {A} {Z} = p
par-subst-zero {M} {M′} p {A} {S x} = pvar
```

We conclude this section with the desired corollary, that substitution respects parallel reduction.

```
sub-par : ∀{Γ A B} {N N′ : Γ , A ⊢ B} {M M′ : Γ ⊢ A}
  → N ⇛ N′
  → M ⇛ M′
    --------------------------
  → N [ M ] ⇛ N′ [ M′ ]
sub-par pn pm = subst-par (par-subst-zero pm) pn
```

Parallel reduction satisfies the diamond property

The heart of the confluence proof is made of stone, or rather, of diamond! We show that parallel reduction satisfies the diamond property: that if `M ⇛ N` and `M ⇛ N′`, then `N ⇛ L` and `N′ ⇛ L` for some `L`. The typical proof is an induction on `M ⇛ N` and `M ⇛ N′` so that every possible pair gives rise to a witness `L` given by performing enough beta reductions in parallel.

However, a simpler approach is to perform as many beta reductions in parallel as possible on `M`, say `M ⁺`, and then show that `N` also parallel reduces to `M ⁺`. This is the idea of Takahashi's *complete development*. The desired property may be illustrated as

where downward lines are instances of `⇛`, so we call it the *triangle property*.

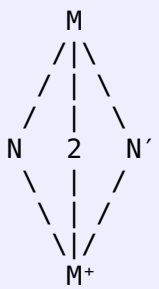
```
_⁺ : ∀ {Γ A}
  → Γ ⊢ A → Γ ⊢ A
(` x) ⁺   = ` x
(ƛ M) ⁺   = ƛ (M ⁺)
((ƛ N) · M) ⁺ = N ⁺ [ M ⁺ ]
(L · M) ⁺ = L ⁺ · (M ⁺)

par-triangle : ∀ {Γ A} {M N : Γ ⊢ A}
  → M ⇛ N
    -------
  → N ⇛ M ⁺
par-triangle pvar     = pvar
par-triangle (pabs p) = pabs (par-triangle p)
par-triangle (pbeta p1 p2) = sub-par (par-triangle p1) (par-triangle p2)
par-triangle (papp {L = ƛ _} (pabs p1) p2) =
  pbeta (par-triangle p1) (par-triangle p2)
par-triangle (papp {L = ` _} p1 p2) = papp (par-triangle p1) (par-triangle p2)
par-triangle (papp {L = _ · _} p1 p2) = papp (par-triangle p1) (par-triangle p2)
```

The proof of the triangle property is an induction on `M ⇛ N`.

- Suppose `x ⇛ x`. Clearly `x ⁺ = x`, so `x ⇛ x`.

- Suppose `ƛ M ⇒ ƛ N`. By the induction hypothesis we have `N ⇒ M ⁺` and by definition `(ƛ M) ⁺ = ƛ (M ⁺)`, so we conclude that `ƛ N ⇒ ƛ (M ⁺)`.
- Suppose `(ƛ N) · M ⇒ N′ [ M′ ]`. By the induction hypothesis, we have `N′ ⇒ N ⁺` and `M′ ⇒ M ⁺`. Since substitution respects parallel reduction, it follows that `N′ [ M′ ] ⇒ N ⁺ [ M ⁺ ]`, but the right hand side is exactly `((ƛ N) · M) ⁺`, hence `N′ [ M′ ] ⇒ ((ƛ N) · M) ⁺`.
- Suppose `(ƛ L) · M ⇒ (ƛ L′) · M′`. By the induction hypothesis we have `L′ ⇒ L ⁺` and `M′ ⇒ M ⁺`; by definition `((ƛ L) · M) ⁺ = L ⁺ [ M ⁺ ]`. It follows `(ƛ L′) · M′ ⇒ L ⁺ [ M ⁺ ]`.
- Suppose `x · M ⇒ x · M′`. By the induction hypothesis we have `M′ ⇒ M ⁺` and `x ⇒ x ⁺` so that `x · M′ ⇒ x · M ⁺`. The remaining case is proved in the same way, so we ignore it. (As there is currently no way in Agda to expand the catch-all pattern in the definition of `_⁺` for us before checking the right-hand side, we have to write down the remaining case explicitly.)

The diamond property then follows by halving the diamond into two triangles.



That is, the diamond property is proved by applying the triangle property on each side with the same confluent term `M ⁺`.

```
par-diamond : ∀{Γ A} {M N N′ : Γ ⊢ A}
  → M ⇒ N
  → M ⇒ N′
    ---------------------------------
  → Σ[ L ∈ Γ ⊢ A ] (N ⇒ L) × (N′ ⇒ L)
par-diamond {M = M} p1 p2 = ⟨ M ⁺ , ⟨ par-triangle p1 , par-triangle p2 ⟩ ⟩
```

This step is optional, though, in the presence of triangle property.

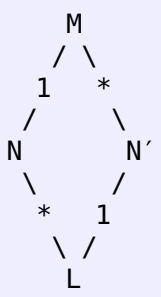
Exercise (practice)

- Prove the diamond property `par-diamond` directly by induction on `M ⇒ N` and `M ⇒ N′`.
- Draw pictures that represent the proofs of each of the six cases in the direct proof of `par-diamond`. The pictures should consist of nodes and directed edges, where each node is labeled with a term and each edge represents parallel reduction.

Proof of confluence for parallel reduction

As promised at the beginning, the proof that parallel reduction is confluent is easy now that we know it satisfies the triangle property. We just need to prove the strip lemma, which states that

if `M ⇒ N` and `M ⇒* N′`, then `N ⇒* L` and `N′ ⇒ L` for some `L`. The following diagram illustrates the strip lemma



where downward lines are instances of `⇒` or `⇒*`, depending on how they are marked.

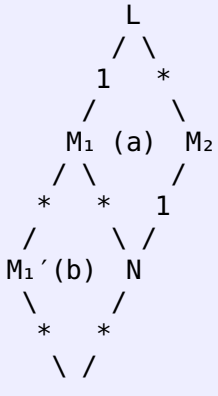
The proof of the strip lemma is a straightforward induction on `M ⇒* N′`, using the triangle property in the induction step.

```
strip : ∀{Γ A} {M N N′ : Γ ⊢ A}
  → M ⇒ N
  → M ⇒* N′
    ------------------------------------
  → Σ[ L ∈ Γ ⊢ A ] (N ⇒* L) × (N′ ⇒ L)
strip{Γ}{A}{M}{N}{N′} mn (M ∎) = ⟨ N , ⟨ N ∎ , mn ⟩ ⟩
strip{Γ}{A}{M}{N}{N′} mn (M ⇒⟨ mm' ⟩ m'n')
  with strip (par-triangle mm') m'n'
... | ⟨ L , ⟨ ll' , n'l' ⟩ ⟩ = ⟨ L , ⟨ N ⇒⟨ par-triangle mn ⟩ ll' , n'l' ⟩ ⟩
```

The proof of confluence for parallel reduction is now proved by induction on the sequence `M ⇒* N`, using the above lemma in the induction step.

```
par-confluence : ∀{Γ A} {L M₁ M₂ : Γ ⊢ A}
  → L ⇒* M₁
  → L ⇒* M₂
    ------------------------------------
  → Σ[ N ∈ Γ ⊢ A ] (M₁ ⇒* N) × (M₂ ⇒* N)
par-confluence {Γ}{A}{L}{.L}{N} (L ∎) L⇒*N = ⟨ N , ⟨ L⇒*N , N ∎ ⟩ ⟩
par-confluence {Γ}{A}{L}{M₁′}{M₂} (L ⇒⟨ L⇒M₁ ⟩ M₁⇒*M₁′) L⇒*M₂
   with strip L⇒M₁ L⇒*M₂
... | ⟨ N , ⟨ M₁⇒*N , M₂⇒N ⟩ ⟩
      with par-confluence M₁⇒*M₁′ M₁⇒*N
...   | ⟨ N′ , ⟨ M₁′⇒*N′ , N⇒*N′ ⟩ ⟩ =
        ⟨ N′ , ⟨ M₁′⇒*N′ , (M₂ ⇒⟨ M₂⇒N ⟩ N⇒*N′) ⟩ ⟩
```

The step case may be illustrated as follows:



```
    N′
```

where downward lines are instances of `⇒` or `⇒*`, depending on how they are marked. Here `(a)` holds by `strip` and `(b)` holds by induction.

Proof of confluence for reduction

Confluence of reduction is a corollary of confluence for parallel reduction. From `L ⟶↠ M₁` and `L ⟶↠ M₂` we have `L ⇒* M₁` and `L ⇒* M₂` by `betas-pars`. Then by confluence we obtain some `L` such that `M₁ ⇒* N` and `M₂ ⇒* N`, from which we conclude that `M₁ ⟶↠ N` and `M₂ ⟶↠ N` by `pars-betas`.

```
confluence : ∀{Γ A} {L M₁ M₂ : Γ ⊢ A}
  → L ⟶↠ M₁
  → L ⟶↠ M₂
    ----------------------------------
  → Σ[ N ∈ Γ ⊢ A ] (M₁ ⟶↠ N) × (M₂ ⟶↠ N)
confluence L↠M₁ L↠M₂
   with par-confluence (betas-pars L↠M₁) (betas-pars L↠M₂)
... | ⟨ N , ⟨ M₁⇒N , M₂⇒N ⟩ ⟩ =
      ⟨ N , ⟨ pars-betas M₁⇒N , pars-betas M₂⇒N ⟩ ⟩
```

Notes

Broadly speaking, this proof of confluence, based on parallel reduction, is due to W. Tait and P. Martin-Löf (see Barendregt 1984, Section 3.2). Details of the mechanization come from several sources. The `subst-par` lemma is the "strong substitutivity" lemma of Shafer, Tebbi, and Smolka (ITP 2015). The proofs of `par-triangle`, `strip`, and `par-confluence` are based on the notion of complete development by Takahashi (1995) and Pfenning's 1992 technical report about the Church-Rosser theorem. In addition, we consulted Nipkow and Berghofer's mechanization in Isabelle, which is based on an earlier article by Nipkow (JAR 1996).

Unicode

This chapter uses the following unicode:

```
⇒  U+21DB  RIGHTWARDS TRIPLE ARROW (\r== or \Rrightarrow)
⁺  U+207A  SUPERSCRIPT PLUS SIGN   (\^+)
```

Chapter 19

BigStep: Big-step semantics of untyped lambda calculus

```
module plfa.part2.BigStep where
```

Introduction

The call-by-name evaluation strategy is a deterministic method for computing the value of a program in the lambda calculus. That is, call-by-name produces a value if and only if beta reduction can reduce the program to a lambda abstraction. In this chapter we define call-by-name evaluation and prove the forward direction of this if-and-only-if. The backward direction is traditionally proved via Curry-Feys standardisation, which is quite complex. We give a sketch of that proof, due to Plotkin, but postpone the proof in Agda until after we have developed a denotational semantics for the lambda calculus, at which point the proof is an easy corollary of properties of the denotational semantics.

We present the call-by-name strategy as a relation between an input term and an output value. Such a relation is often called a *big-step semantics*, written `M ⇓ V`, as it relates the input term `M` directly to the final result `V`, in contrast to the small-step reduction relation, `M ⟶ M′`, that maps `M` to another term `M′` in which a single sub-computation has been completed.

Imports

```
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; trans; sym; cong-app)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Function using (_∘_)
open import plfa.part2.Untyped
  using (Context; _⊢_; _∋_; ★; ∅; _,_; Z; S_; `_; #_; ƛ_; _·_;
  subst; subst-zero; exts; rename; β; ξ₁; ξ₂; ζ; _⟶_; _⟶↠_; _⟶⟨_⟩_; _∎;
  ⟶↠-trans; appL-cong)
open import plfa.part2.Substitution using (Subst; ids)
```

Environments

To handle variables and function application, there is the choice between using substitution, as in `⟶`, or to use an *environment*. An environment in call-by-name is a map from variables to closures, that is, to terms paired with their environments. We choose to use environments instead of substitution because the point of the call-by-name strategy is to be closer to an implementation of the language. Also, the denotational semantics introduced in later chapters uses environments and the proof of adequacy is made easier by aligning these choices.

We define environments and closures as follows.

```
ClosEnv : Context → Set

data Clos : Set where
  clos : ∀{Γ} → (M : Γ ⊢ ★) → ClosEnv Γ → Clos

ClosEnv Γ = ∀ (x : Γ ∋ ★) → Clos
```

As usual, we have the empty environment, and we can extend an environment.

```
∅' : ClosEnv ∅
∅' ()

_,'_ : ∀ {Γ} → ClosEnv Γ → Clos → ClosEnv (Γ , ★)
(γ ,' c) Z = c
(γ ,' c) (S x) = γ x
```

Big-step evaluation

The big-step semantics is represented as a ternary relation, written `γ ⊢ M ⇓ V`, where `γ` is the environment, `M` is the input term, and `V` is the result value. A *value* is a closure whose term is a lambda abstraction.

```
data _⊢_⇓_ : ∀{Γ} → ClosEnv Γ → (Γ ⊢ ★) → Clos → Set where

  ⇓-var : ∀{Γ}{γ : ClosEnv Γ}{x : Γ ∋ ★}{Δ}{δ : ClosEnv Δ}{M : Δ ⊢ ★}{V}
    → γ x ≡ clos M δ
    → δ ⊢ M ⇓ V
      -----------
    → γ ⊢ ` x ⇓ V

  ⇓-lam : ∀{Γ}{γ : ClosEnv Γ}{M : Γ , ★ ⊢ ★}
    → γ ⊢ ƛ M ⇓ clos (ƛ M) γ

  ⇓-app : ∀{Γ}{γ : ClosEnv Γ}{L M : Γ ⊢ ★}{Δ}{δ : ClosEnv Δ}{N : Δ , ★ ⊢ ★}{V}
    → γ ⊢ L ⇓ clos (ƛ N) δ → (δ ,' clos M γ) ⊢ N ⇓ V
      ---------------------------------------------
    → γ ⊢ L · M ⇓ V
```

- The `⇓-var` rule evaluates a variable by finding the associated closure in the environment and then evaluating the closure.
- The `⇓-lam` rule turns a lambda abstraction into a closure by packaging it up with its environment.

- The `⇓-app` rule performs function application by first evaluating the term `L` in operator position. If that produces a closure containing a lambda abstraction `ƛ N`, then we evaluate the body `N` in an environment extended with the argument `M`. Note that `M` is not evaluated in rule `⇓-app` because this is call-by-name and not call-by-value.

Exercise `big-step-eg` (practice)

Show that `(ƛ ƛ # 1) · ((ƛ # 0 · # 0) · (ƛ # 0 · # 0))` terminates under big-step call-by-name evaluation.

```
-- Your code goes here
```

The big-step semantics is deterministic

If the big-step relation evaluates a term `M` to both `V` and `V′`, then `V` and `V′` must be identical. In other words, the call-by-name relation is a partial function. The proof is a straightforward induction on the two big-step derivations.

```
⇓-determ : ∀{Γ}{γ : ClosEnv Γ}{M : Γ ⊢ ★}{V V' : Clos}
  → γ ⊢ M ⇓ V → γ ⊢ M ⇓ V'
  → V ≡ V'
⇓-determ (⇓-var eq1 mc) (⇓-var eq2 mc')
  with trans (sym eq1) eq2
... | refl = ⇓-determ mc mc'
⇓-determ ⇓-lam ⇓-lam = refl
⇓-determ (⇓-app mc mc₁) (⇓-app mc' mc'')
  with ⇓-determ mc mc'
... | refl = ⇓-determ mc₁ mc''
```

Big-step evaluation implies beta reduction to a lambda

If big-step evaluation produces a value, then the input term can reduce to a lambda abstraction by beta reduction:

```
  ∅' ⊢ M ⇓ clos (ƛ N′) δ
  ----------------------------
→ Σ[ N ∈ ∅ , ★ ⊢ ★ ] (M —↠ ƛ N)
```

The proof is by induction on the big-step derivation. As is often necessary, one must generalize the statement to get the induction to go through. In the case for `⇓-app` (function application), the argument is added to the environment, so the environment becomes non-empty. The corresponding β reduction substitutes the argument into the body of the lambda abstraction. So we generalize the lemma to allow an arbitrary environment `γ` and we add a premise that relates the environment `γ` to an equivalent substitution `σ`.

The case for `⇓-app` also requires that we strengthen the conclusion. In the case for `⇓-app` we have `γ ⊢ L ⇓ clos (λ N) δ` and the induction hypothesis gives us `L —↠ ƛ N′`, but we need to know that `N` and `N′` are equivalent. In particular, that `N′ ≡ subst τ N` where `τ` is the

substitution that is equivalent to `δ`. Therefore we expand the conclusion of the statement, stating that the results are equivalent.

We make the two notions of equivalence precise by defining the following two mutually-recursive predicates `V ≈ M` and `γ ≈ₑ σ`.

```
_≈_ : Clos → (∅ ⊢ ★) → Set
_≈ₑ_ : ∀{Γ} → ClosEnv Γ → Subst Γ ∅ → Set

(clos {Γ} M γ) ≈ N = Σ[ σ ∈ Subst Γ ∅ ] γ ≈ₑ σ × (N ≡ subst σ M)

γ ≈ₑ σ = ∀{x} → (γ x) ≈ (σ x)
```

We can now state the main lemma:

```
If γ ⊢ M ⇓ V  and  γ ≈ₑ σ,
then  subst σ M —↠ N  and  V ≈ N  for some N.
```

Before starting the proof, we establish a couple lemmas about equivalent environments and substitutions.

The empty environment is equivalent to the identity substitution `ids`, which we import from Chapter Substitution.

```
≈ₑ-id : ∅' ≈ₑ ids
≈ₑ-id {()}
```

Of course, applying the identity substitution to a term returns the same term.

```
sub-id : ∀{Γ} {A} {M : Γ ⊢ A} → subst ids M ≡ M
sub-id = plfa.part2.Substitution.sub-id
```

We define an auxiliary function for extending a substitution.

```
ext-subst : ∀{Γ Δ} → Subst Γ Δ → Δ ⊢ ★ → Subst (Γ , ★) Δ
ext-subst{Γ}{Δ} σ N {A} = subst (subst-zero N) ∘ exts σ
```

The next lemma we need to prove states that if you start with an equivalent environment and substitution `γ ≈ₑ σ`, extending them with an equivalent closure and term `c ≈ N` produces an equivalent environment and substitution: `(γ ,' V) ≈ₑ (ext-subst σ N)`, or equivalently, `(γ ,' V) x ≈ₑ (ext-subst σ N) x` for any variable `x`. The proof will be by induction on `x` and for the induction step we need the following lemma, which states that applying the composition of `exts σ` and `subst-zero` to `S x` is the same as just `σ x`, which is a corollary of a theorem in Chapter Substitution.

```
subst-zero-exts : ∀{Γ Δ}{σ : Subst Γ Δ}{B}{M : Δ ⊢ B}{x : Γ ∋ ★}
  → (subst (subst-zero M) ∘ exts σ) (S x) ≡ σ x
subst-zero-exts {Γ}{Δ}{σ}{B}{M}{x} =
  cong-app (plfa.part2.Substitution.subst-zero-exts-cons{σ = σ}) (S x)
```

So the proof of `≈ₑ-ext` is as follows.

```
≈ₑ-ext : ∀ {Γ} {γ : ClosEnv Γ} {σ : Subst Γ ∅} {V} {N : ∅ ⊢ ★}
  → γ ≈ₑ σ → V ≈ N
    -------------------------
  → (γ ,' V) ≈ₑ (ext-subst σ N)
```

```
≈ₑ-ext {Γ} {γ} {σ} {V} {N} γ≈ₑσ V≈N {Z} = V≈N
≈ₑ-ext {Γ} {γ} {σ} {V} {N} γ≈ₑσ V≈N {S x}
  rewrite subst-zero-exts {σ = σ}{M = N}{x} = γ≈ₑσ
```

We proceed by induction on the input variable.

- If it is `Z`, then we immediately conclude using the premise `V ≈ N`.
- If it is `S x`, then we rewrite using the `subst-zero-exts` lemma and use the premise `γ ≈ₑ σ` to conclude.

To prove the main lemma, we need another technical lemma about substitution. Applying one substitution after another is the same as composing the two substitutions and then applying them.

```
sub-sub : ∀{Γ Δ Σ}{A}{M : Γ ⊢ A} {σ₁ : Subst Γ Δ}{σ₂ : Subst Δ Σ}
  → subst σ₂ (subst σ₁ M) ≡ subst (subst σ₂ ∘ σ₁) M
sub-sub {M = M} = plfa.part2.Substitution.sub-sub {M = M}
```

We arive at the main lemma: if `M` big steps to a closure `V` in environment `γ`, and if `γ ≈ₑ σ`, then `subst σ M` reduces to some term `N` that is equivalent to `V`. We describe the proof below.

```
⇓→⟶→x≈ : ∀{Γ}{γ : ClosEnv Γ}{σ : Subst Γ ∅}{M : Γ ⊢ ⋆}{V : Clos}
       → γ ⊢ M ⇓ V → γ ≈ₑ σ
         ---------------------------------------
       → Σ[ N ∈ ∅ ⊢ ⋆ ] (subst σ M ⟶↠ N) × V ≈ N
⇓→⟶→x≈ {γ = γ} (⇓-var{x = x} γx≡Lδ δ⊢L⇓V) γ≈ₑσ
  with γ x | γ≈ₑσ {x} | γx≡Lδ
... | clos L δ | ⟨ τ , ⟨ δ≈ₑτ , σx≡τL ⟩ ⟩ | refl
      with ⇓→⟶→x≈{σ = τ} δ⊢L⇓V δ≈ₑτ
...   | ⟨ N , ⟨ τL⟶↠N , V≈N ⟩ ⟩ rewrite σx≡τL =
        ⟨ N , ⟨ τL⟶↠N , V≈N ⟩ ⟩
⇓→⟶→x≈ {σ = σ} {V = clos (ƛ N) γ} (⇓-lam) γ≈ₑσ =
  ⟨ subst σ (ƛ N) , ⟨ subst σ (ƛ N) ∎ , ⟨ σ , ⟨ γ≈ₑσ , refl ⟩ ⟩ ⟩ ⟩
⇓→⟶→x≈{Γ}{γ} {σ = σ} {L · M} {V} (⇓-app {N = N} L⇓ƛNδ N⇓V) γ≈ₑσ
  with ⇓→⟶→x≈{σ = σ} L⇓ƛNδ γ≈ₑσ
... | ⟨ _ , ⟨ σL⟶↠ƛτN , ⟨ τ , ⟨ δ≈ₑτ , ≡ƛτN ⟩ ⟩ ⟩ ⟩ rewrite ≡ƛτN
      with ⇓→⟶→x≈ {σ = ext-subst τ (subst σ M)} N⇓V
             (λ {x} → ≈ₑ-ext{σ = τ} δ≈ₑτ ⟨ σ , ⟨ γ≈ₑσ , refl ⟩ ⟩ {x})
         | β{∅}{subst (exts τ) N}{subst σ M}
...   | ⟨ N' , ⟨ ⟶↠N' , V≈N' ⟩ ⟩ | ƛτN·σM⟶
        rewrite sub-sub{M = N}{σ₁ = exts τ}{σ₂ = subst-zero (subst σ M)} =
        let rs = (ƛ subst (exts τ) N) · subst σ M ⟶⟨ ƛτN·σM⟶ ⟩ ⟶↠N' in
        let g = ⟶↠-trans (appL-cong σL⟶↠ƛτN) rs in
        ⟨ N' , ⟨ g , V≈N' ⟩ ⟩
```

The proof is by induction on `γ ⊢ M ⇓ V`. We have three cases to consider.

- Case `⇓-var`. So we have `γ x ≡ clos L δ` and `δ ⊢ L ⇓ V`. We need to show that `subst σ x ⟶↠ N` and `V ≈ N` for some `N`. The premise `γ ≈ₑ σ` tells us that `γ x ≈ σ x`, so `clos L δ ≈ σ x`. By the definition of `≈`, there exists a `τ` such that `δ ≈ₑ τ` and `σ x ≡ subst τ L`. Using `δ ⊢ L ⇓ V` and `δ ≈ₑ τ`, the induction hypothesis gives us `subst τ L ⟶↠ N` and `V ≈ N` for some `N`. So we have shown that `subst σ x ⟶↠ N` and `V ≈ N` for some `N`.

- Case `⇓-lam`. We immediately have `subst σ (ƛ N) —↠ subst σ (ƛ N)` and `clos (subst σ (ƛ N)) γ ≈ subst σ (ƛ N)`.
- Case `⇓-app`. Using `γ ⊢ L ⇓ clos N δ` and `γ ≈ₑ σ`, the induction hypothesis gives us

  ```
  subst σ L —↠ ƛ subst (exts τ) N                                    (1)
  ```

  and `δ ≈ₑ τ` for some `τ`. From `γ≈ₑσ` we have `clos M γ ≈ subst σ M`. Then with `(δ ,' clos M γ) ⊢ N ⇓ V`, the induction hypothesis gives us `V ≈ N'` and

  ```
  subst (subst (subst-zero (subst σ M)) ∘ (exts τ)) N —↠ N'          (2)
  ```

  Meanwhile, by `β`, we have

  ```
  (ƛ subst (exts τ) N) · subst σ M
  —→ subst (subst-zero (subst σ M)) (subst (exts τ) N)
  ```

  which is the same as the following, by `sub-sub`.

  ```
  (ƛ subst (exts τ) N) · subst σ M
  —→ subst (subst (subst-zero (subst σ M)) ∘ exts τ) N               (3)
  ```

  Using (3) and (2) we have

  ```
  (ƛ subst (exts τ) N) · subst σ M —↠ N'                              (4)
  ```

  From (1) we have

  ```
  subst σ L · subst σ M —↠ (ƛ subst (exts τ) N) · subst σ M
  ```

  which we combine with (4) to conclude that

  ```
  subst σ L · subst σ M —↠ N'
  ```

With the main lemma complete, we establish the forward direction of the equivalence between the big-step semantics and beta reduction.

```
cbn→reduce : ∀{M : ∅ ⊢ ★}{Δ}{δ : ClosEnv Δ}{N′ : Δ , ★ ⊢ ★}
  → ∅' ⊢ M ⇓ clos (ƛ N′) δ
    ----------------------------
  → Σ[ N ∈ ∅ , ★ ⊢ ★ ] (M —↠ ƛ N)
cbn→reduce {M}{Δ}{δ}{N′} M⇓c
   with ⇓→—↠×≈{σ = ids} M⇓c ≈ₑ-id
... | ⟨ N , ⟨ rs , ⟨ σ , ⟨ h , eq2 ⟩ ⟩ ⟩ ⟩ rewrite sub-id{M = M} | eq2 =
      ⟨ subst (exts σ) N′ , rs ⟩
```

Exercise `big-alt-implies-multi` **(practice)**

Formulate an alternative big-step semantics, of the form `M ↓ N`, for call-by-name that uses substitution instead of environments. That is, the analogue of the application rule `⇓-app` should perform substitution, as in `N [ M ]`, instead of extending the environment with `M`. Prove that `M ↓ N` implies `M —↠ N`.

```
-- Your code goes here
```

Beta reduction to a lambda implies big-step evaluation

The proof of the backward direction, that beta reduction to a lambda implies that the call-by-name semantics produces a result, is more difficult to prove. The difficulty stems from reduction proceeding underneath lambda abstractions via the `ζ` rule. The call-by-name semantics does not reduce under lambda, so a straightforward proof by induction on the reduction sequence is impossible. In the article *Call-by-name, call-by-value, and the λ-calculus*, Plotkin proves the theorem in two steps, using two auxiliary reduction relations. The first step uses a classic technique called Curry-Feys standardisation. It relies on the notion of *standard reduction sequence*, which acts as a half-way point between full beta reduction and call-by-name by expanding call-by-name to also include reduction underneath lambda. Plotkin proves that `M` reduces to `L` if and only if `M` is related to `L` by a standard reduction sequence.

```
Theorem 1 (Standardisation)
`M —↠ L` if and only if `M` goes to `L` via a standard reduction sequence.
```

Plotkin then introduces *left reduction*, a small-step version of call-by-name and uses the above theorem to prove that beta reduction and left reduction are equivalent in the following sense.

```
Corollary 1
`M —↠ ƛ N` if and only if `M` goes to `ƛ N′`, for some `N′`, by left reduction.
```

The second step of the proof connects left reduction to call-by-name evaluation.

```
Theorem 2
`M` left reduces to `ƛ N` if and only if `⊢ M ⇓ ƛ N`.
```

(Plotkin's call-by-name evaluator uses substitution instead of environments, which explains why the environment is omitted in `⊢ M ⇓ ƛ N` in the above theorem statement.)

Putting Corollary 1 and Theorem 2 together, Plotkin proves that call-by-name evaluation is equivalent to beta reduction.

```
Corollary 2
`M —↠ ƛ N` if and only if `⊢ M ⇓ ƛ N′` for some `N′`.
```

Plotkin also proves an analogous result for the λ_v calculus, relating it to call-by-value evaluation. For a nice exposition of that proof, we recommend Chapter 5 of *Semantics Engineering with PLT Redex* by Felleisen, Findler, and Flatt.

Instead of proving the backwards direction via standardisation, as sketched above, we defer the proof until after we define a denotational semantics for the lambda calculus, at which point the proof of the backwards direction will fall out as a corollary to the soundness and adequacy of the denotational semantics.

Unicode

This chapter uses the following unicode:

```
≈  U+2248  ALMOST EQUAL TO (\~~ or \approx)
ₑ  U+2091  LATIN SUBSCRIPT SMALL LETTER E (\_e)
⊢  U+22A2  RIGHT TACK (\|- or \vdash)
⇓  U+21DB  DOWNWARDS DOUBLE ARROW (\d= or \Downarrow)
```

Part III

Part 3: Denotational Semantics

Chapter 20

Denotational: Denotational semantics of untyped lambda calculus

```
module plfa.part3.Denotational where
```

The lambda calculus is a language about *functions*, that is, mappings from input to output. In computing we often think of such mappings as being carried out by a sequence of operations that transform an input into an output. But functions can also be represented as data. For example, one can tabulate a function, that is, create a table where each row has two entries, an input and the corresponding output for the function. Function application is then the process of looking up the row for a given input and reading off the output.

We shall create a semantics for the untyped lambda calculus based on this idea of functions-as-tables. However, there are two difficulties that arise. First, functions often have an infinite domain, so it would seem that we would need infinitely long tables to represent functions. Second, in the lambda calculus, functions can be applied to functions. They can even be applied to themselves! So it would seem that the tables would contain cycles. One might start to worry that advanced techniques are necessary to address these issues, but fortunately this is not the case!

The first problem, of functions with infinite domains, is solved by observing that in the execution of a terminating program, each lambda abstraction will only be applied to a finite number of distinct arguments. (We come back later to discuss diverging programs.) This observation is another way of looking at Dana Scott's insight that only continuous functions are needed to model the lambda calculus.

The second problem, that of self-application, is solved by relaxing the way in which we lookup an argument in a function's table. Naively, one would look in the table for a row in which the input entry exactly matches the argument. In the case of self-application, this would require the table to contain a copy of itself. Impossible! (At least, it is impossible if we want to build tables using inductive data type definitions, which indeed we do.) Instead it is sufficient to find an input such that every row of the input appears as a row of the argument (that is, the input is a subset of the argument). In the case of self-application, the table only needs to contain a smaller copy of itself, which is fine.

With these two observations in hand, it is straightforward to write down a denotational semantics of the lambda calculus.

Imports

```
open import Agda.Primitive using (lzero; lsuc)
open import Data.Empty using (⊥-elim)
open import Data.Nat using (ℕ; zero; suc)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Data.Sum
open import Data.Vec using (Vec; []; _∷_)
open import Relation.Binary.PropositionalEquality
  using (_≡_; _≢_; refl; sym; cong; cong₂; cong-app)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Negation using (contradiction)
open import Function using (_∘_)
open import plfa.part2.Untyped
  using (Context; ★; _∋_; ∅; _,_; Z; S_; _⊢_; `_; _·_; ƛ_;
         #_; two ᶜ; ext; rename; exts; subst; subst-zero; _[_])
open import plfa.part2.Substitution using (Rename; extensionality; rename-id)
```

Values

The `Value` data type represents a finite portion of a function. We think of a value as a finite set of pairs that represent input-output mappings. The `Value` data type represents the set as a binary tree whose internal nodes are the union operator and whose leaves represent either a single mapping or the empty set.

- The ⊥ value provides no information about the computation.
- A value of the form `v ↦ w` is a single input-output mapping, from input `v` to output `w`.
- A value of the form `v ⊔ w` is a function that maps inputs to outputs according to both `v` and `w`. Think of it as taking the union of the two sets.

```
infixr 7 _↦_
infixl 5 _⊔_

data Value : Set where
  ⊥ : Value
  _↦_ : Value → Value → Value
  _⊔_ : Value → Value → Value
```

The `⊑` relation adapts the familiar notion of subset to the Value data type. This relation plays the key role in enabling self-application. There are two rules that are specific to functions, `⊑-fun` and `⊑-dist`, which we discuss below.

```
infix 4 _⊑_

data _⊑_ : Value → Value → Set where

  ⊑-bot : ∀ {v} → ⊥ ⊑ v

  ⊑-conj-L : ∀ {u v w}
    → v ⊑ u
    → w ⊑ u
```

```
    -----------
  → (v ⊔ w) ⊑ u

⊑-conj-R1 : ∀ {u v w}
  → u ⊑ v
    -----------
  → u ⊑ (v ⊔ w)

⊑-conj-R2 : ∀ {u v w}
  → u ⊑ w
    -----------
  → u ⊑ (v ⊔ w)

⊑-trans : ∀ {u v w}
  → u ⊑ v
  → v ⊑ w
    -----
  → u ⊑ w

⊑-fun : ∀ {v w v′ w′}
  → v′ ⊑ v
  → w ⊑ w′
    --------------------
  → (v ↦ w) ⊑ (v′ ↦ w′)

⊑-dist : ∀{v w w′}
    ---------------------------------
  → v ↦ (w ⊔ w′) ⊑ (v ↦ w) ⊔ (v ↦ w′)
```

The first five rules are straightforward. The rule `⊑-fun` captures when it is OK to match a higher-order argument `v′ ↦ w′` to a table entry whose input is `v ↦ w`. Considering a call to the higher-order argument. It is OK to pass a larger argument than expected, so `v` can be larger than `v′`. Also, it is OK to disregard some of the output, so `w` can be smaller than `w′`. The rule `⊑-dist` says that if you have two entries for the same input, then you can combine them into a single entry and joins the two outputs.

The `⊑` relation is reflexive.

```
⊑-refl : ∀ {v} → v ⊑ v
⊑-refl {⊥} = ⊑-bot
⊑-refl {v ↦ v′} = ⊑-fun ⊑-refl ⊑-refl
⊑-refl {v₁ ⊔ v₂} = ⊑-conj-L (⊑-conj-R1 ⊑-refl) (⊑-conj-R2 ⊑-refl)
```

The `⊔` operation is monotonic with respect to `⊑`, that is, given two larger values it produces a larger value.

```
⊔⊑⊔ : ∀ {v w v′ w′}
  → v ⊑ v′ → w ⊑ w′
    ------------------------
  → (v ⊔ w) ⊑ (v′ ⊔ w′)
⊔⊑⊔ d₁ d₂ = ⊑-conj-L (⊑-conj-R1 d₁) (⊑-conj-R2 d₂)
```

The `⊑-dist` rule can be used to combine two entries even when the input values are not identical. One can first combine the two inputs using `⊔` and then apply the `⊑-dist` rule to obtain the following property.

```
⊔↦⊔-dist : ∀{v v′ w w′ : Value}
  → (v ⊔ v′) ↦ (w ⊔ w′) ⊑ (v ↦ w) ⊔ (v′ ↦ w′)
⊔↦⊔-dist = ⊑-trans ⊑-dist (⊔⊑⊔ (⊑-fun (⊑-conj-R1 ⊑-refl) ⊑-refl)
                           (⊑-fun (⊑-conj-R2 ⊑-refl) ⊑-refl))
```

If the join `u ⊔ v` is less than another value `w`, then both `u` and `v` are less than `w`.

```
⊔⊑-invL : ∀{u v w : Value}
  → u ⊔ v ⊑ w
    ---------
  → u ⊑ w
⊔⊑-invL (⊑-conj-L lt1 lt2) = lt1
⊔⊑-invL (⊑-conj-R1 lt) = ⊑-conj-R1 (⊔⊑-invL lt)
⊔⊑-invL (⊑-conj-R2 lt) = ⊑-conj-R2 (⊔⊑-invL lt)
⊔⊑-invL (⊑-trans lt1 lt2) = ⊑-trans (⊔⊑-invL lt1) lt2

⊔⊑-invR : ∀{u v w : Value}
  → u ⊔ v ⊑ w
    ---------
  → v ⊑ w
⊔⊑-invR (⊑-conj-L lt1 lt2) = lt2
⊔⊑-invR (⊑-conj-R1 lt) = ⊑-conj-R1 (⊔⊑-invR lt)
⊔⊑-invR (⊑-conj-R2 lt) = ⊑-conj-R2 (⊔⊑-invR lt)
⊔⊑-invR (⊑-trans lt1 lt2) = ⊑-trans (⊔⊑-invR lt1) lt2
```

Environments

An environment gives meaning to the free variables in a term by mapping variables to values.

```
Env : Context → Set
Env Γ = ∀ (x : Γ ∋ ⋆) → Value
```

We have the empty environment, and we can extend an environment.

```
`∅ : Env ∅
`∅ ()

infixl 5 _`,_

_`,_ : ∀ {Γ} → Env Γ → Value → Env (Γ , ⋆)
(γ `, v) Z = v
(γ `, v) (S x) = γ x
```

We can recover the previous environment from an extended environment, and the last value. Putting them together again takes us back to where we started.

```
init : ∀ {Γ} → Env (Γ , ⋆) → Env Γ
init γ x = γ (S x)

last : ∀ {Γ} → Env (Γ , ⋆) → Value
last γ = γ Z

init-last : ∀ {Γ} → (γ : Env (Γ , ⋆)) → γ ≡ (init γ `, last γ)
init-last {Γ} γ = extensionality lemma
  where lemma : ∀ (x : Γ , ⋆ ∋ ⋆) → γ x ≡ (init γ `, last γ) x
```

```
        lemma Z     = refl
        lemma (S x) = refl
```

We extend the `⊑` relation point-wise to environments with the following definition.

```
_`⊑_ : ∀ {Γ} → Env Γ → Env Γ → Set
_`⊑_ {Γ} γ δ = ∀ (x : Γ ∋ ⋆) → γ x ⊑ δ x
```

We define a bottom environment and a join operator on environments, which takes the point-wise join of their values.

```
`⊥ : ∀ {Γ} → Env Γ
`⊥ x = ⊥

_`⊔_ : ∀ {Γ} → Env Γ → Env Γ → Env Γ
(γ `⊔ δ) x = γ x ⊔ δ x
```

The `⊑-refl`, `⊑-conj-R1`, and `⊑-conj-R2` rules lift to environments. So the join of two environments `γ` and `δ` is greater than the first environment `γ` or the second environment `δ`.

```
`⊑-refl : ∀ {Γ} {γ : Env Γ} → γ `⊑ γ
`⊑-refl {Γ} {γ} x = ⊑-refl {γ x}

⊑-env-conj-R1 : ∀ {Γ} → (γ : Env Γ) → (δ : Env Γ) → γ `⊑ (γ `⊔ δ)
⊑-env-conj-R1 γ δ x = ⊑-conj-R1 ⊑-refl

⊑-env-conj-R2 : ∀ {Γ} → (γ : Env Γ) → (δ : Env Γ) → δ `⊑ (γ `⊔ δ)
⊑-env-conj-R2 γ δ x = ⊑-conj-R2 ⊑-refl
```

Denotational Semantics

We define the semantics with a judgment of the form `ρ ⊢ M ↓ v`, where `ρ` is the environment, `M` the program, and `v` is a result value. For readers familiar with big-step semantics, this notation will feel quite natural, but don't let the similarity fool you. There are subtle but important differences! So here is the definition of the semantics, which we discuss in detail in the following paragraphs.

```
infix 3 _⊢_↓_

data _⊢_↓_ : ∀{Γ} → Env Γ → (Γ ⊢ ⋆) → Value → Set where

  var : ∀ {Γ} {γ : Env Γ} {x}
      ---------------
    → γ ⊢ (` x) ↓ γ x

  ↦-elim : ∀ {Γ} {γ : Env Γ} {L M v w}
    → γ ⊢ L ↓ (v ↦ w)
    → γ ⊢ M ↓ v
      ---------------
    → γ ⊢ (L · M) ↓ w

  ↦-intro : ∀ {Γ} {γ : Env Γ} {N v w}
    → γ `, v ⊢ N ↓ w
      -------------------
    → γ ⊢ (ƛ N) ↓ (v ↦ w)
```

```
⊥-intro : ∀ {Γ} {γ : Env Γ} {M}
    ---------
  → γ ⊢ M ↓ ⊥

⊔-intro : ∀ {Γ} {γ : Env Γ} {M v w}
  → γ ⊢ M ↓ v
  → γ ⊢ M ↓ w
    ---------------
  → γ ⊢ M ↓ (v ⊔ w)

sub : ∀ {Γ} {γ : Env Γ} {M v w}
  → γ ⊢ M ↓ v
  → w ⊑ v
    ---------
  → γ ⊢ M ↓ w
```

Consider the rule for lambda abstractions, `↦-intro`. It says that a lambda abstraction results in a single-entry table that maps the input `v` to the output `w`, provided that evaluating the body in an environment with `v` bound to its parameter produces the output `w`. As a simple example of this rule, we can see that the identity function maps `⊥` to `⊥` and also that it maps `⊥ ↦ ⊥` to `⊥ ↦ ⊥`.

```
id : ∅ ⊢ ★
id = ƛ # 0
```

```
denot-id1 : ∀ {γ} → γ ⊢ id ↓ ⊥ ↦ ⊥
denot-id1 = ↦-intro var

denot-id2 : ∀ {γ} → γ ⊢ id ↓ (⊥ ↦ ⊥) ↦ (⊥ ↦ ⊥)
denot-id2 = ↦-intro var
```

Of course, we will need tables with many rows to capture the meaning of lambda abstractions. These can be constructed using the `⊔-intro` rule. If term M (typically a lambda abstraction) can produce both tables `v` and `w`, then it produces the combined table `v ⊔ w`. One can take an operational view of the rules `↦-intro` and `⊔-intro` by imagining that when an interpreter first comes to a lambda abstraction, it pre-evaluates the function on a bunch of randomly chosen arguments, using many instances of the rule `↦-intro`, and then joins them into a big table using many instances of the rule `⊔-intro`. In the following we show that the identity function produces a table containing both of the previous results, `⊥ ↦ ⊥` and `(⊥ ↦ ⊥) ↦ (⊥ ↦ ⊥)`.

```
denot-id3 : `∅ ⊢ id ↓ (⊥ ↦ ⊥) ⊔ (⊥ ↦ ⊥) ↦ (⊥ ↦ ⊥)
denot-id3 = ⊔-intro denot-id1 denot-id2
```

We most often think of the judgment `γ ⊢ M ↓ v` as taking the environment `γ` and term `M` as input, producing the result `v`. However, it is worth emphasizing that the semantics is a *relation*. The above results for the identity function show that the same environment and term can be mapped to different results. However, the results for a given `γ` and `M` are not *too* different, they are all finite approximations of the same function. Perhaps a better way of thinking about the judgment `γ ⊢ M ↓ v` is that the `γ`, `M`, and `v` are all inputs and the semantics either confirms or denies whether `v` is an accurate partial description of the result of `M` in environment `γ`.

Next we consider the meaning of function application as given by the `↦-elim` rule. In the premise of the rule we have that `L` maps `v` to `w`. So if `M` produces `v`, then the application of `L` to `M` produces `w`.

As an example of function application and the `↦-elim` rule, we apply the identity function to itself. Indeed, we have both that `∅ ⊢ id ↓ (u ↦ u) ↦ (u ↦ u)` and also `∅ ⊢ id ↓ (u ↦ u)`, so we can apply the rule `↦-elim`.

```
id-app-id : ∀ {u : Value} → `∅ ⊢ id · id ↓ (u ↦ u)
id-app-id {u} = ↦-elim (↦-intro var) (↦-intro var)
```

Next we revisit the Church numeral two: `λ f. λ u. (f (f u))`. This function has two parameters: a function `f` and an arbitrary value `u`, and it applies `f` twice. So `f` must map `u` to some value, which we'll name `v`. Then for the second application, `f` must map `v` to some value. Let's name it `w`. So the function's table must include two entries, both `u ↦ v` and `v ↦ w`. For each application of the table, we extract the appropriate entry from it using the `sub` rule. In particular, we use the ⊑-conj-R1 and ⊑-conj-R2 to select `u ↦ v` and `v ↦ w`, respectively, from the table `u ↦ v ⊔ v ↦ w`. So the meaning of two^c is that it takes this table and parameter `u`, and it returns `w`. Indeed we derive this as follows.

```
denot-two^c : ∀{u v w : Value} → `∅ ⊢ two^c ↓ ((u ↦ v ⊔ v ↦ w) ↦ u ↦ w)
denot-two^c {u}{v}{w} =
  ↦-intro (↦-intro (↦-elim (sub var lt1) (↦-elim (sub var lt2) var)))
  where lt1 : v ↦ w ⊑ u ↦ v ⊔ v ↦ w
        lt1 = ⊑-conj-R2 (⊑-fun ⊑-refl ⊑-refl)

        lt2 : u ↦ v ⊑ u ↦ v ⊔ v ↦ w
        lt2 = (⊑-conj-R1 (⊑-fun ⊑-refl ⊑-refl))
```

Next we have a classic example of self application: `Δ = λx. (x x)`. The input value for `x` needs to be a table, and it needs to have an entry that maps a smaller version of itself, call it `v`, to some value `w`. So the input value looks like `v ↦ w ⊔ v`. Of course, then the output of `Δ` is `w`. The derivation is given below. The first occurrences of `x` evaluates to `v ↦ w`, the second occurrence of `x` evaluates to `v`, and then the result of the application is `w`.

```
Δ : ∅ ⊢ ★
Δ = (ƛ (# 0) · (# 0))

denot-Δ : ∀ {v w} → `∅ ⊢ Δ ↓ ((v ↦ w ⊔ v) ↦ w)
denot-Δ = ↦-intro (↦-elim (sub var (⊑-conj-R1 ⊑-refl))
                          (sub var (⊑-conj-R2 ⊑-refl)))
```

One might worry whether this semantics can deal with diverging programs. The `⊥` value and the `⊥-intro` rule provide a way to handle them. (The `⊥-intro` rule is also what enables β reduction on non-terminating arguments.) The classic `Ω` program is a particularly simple program that diverges. It applies `Δ` to itself. The semantics assigns to `Ω` the meaning `⊥`. There are several ways to derive this, we shall start with one that makes use of the `⊔-intro` rule. First, `denot-Δ` tells us that `Δ` evaluates to `((⊥ ↦ ⊥) ⊔ ⊥) ↦ ⊥` (choose $v_1 = v_2 = \bot$). Next, `Δ` also evaluates to `⊥ ↦ ⊥` by use of `↦-intro` and `⊥-intro` and to `⊥` by `⊥-intro`. As we saw previously, whenever we can show that a program evaluates to two values, we can apply `⊔-intro` to join them together, so `Δ` evaluates to `(⊥ ↦ ⊥) ⊔ ⊥`. This matches the input of the first occurrence of `Δ`, so we can conclude that the result of the application is `⊥`.

```
Ω : ∅ ⊢ ★
Ω = Δ · Δ

denot-Ω : `∅ ⊢ Ω ↓ ⊥
denot-Ω = ↦-elim denot-Δ (⊔-intro (↦-intro ⊥-intro) ⊥-intro)
```

A shorter derivation of the same result is by just one use of the `⊥-intro` rule.

```
denot-Ω' : `∅ ⊢ Ω ↓ ⊥
denot-Ω' = ⊥-intro
```

Just because one can derive `∅ ⊢ M ↓ ⊥` for some closed term `M` doesn't mean that `M` necessarily diverges. There may be other derivations that conclude with `M` producing some more informative value. However, if the only thing that a term evaluates to is `⊥`, then it indeed diverges.

An attentive reader may have noticed a disconnect earlier in the way we planned to solve the self-application problem and the actual `↦-elim` rule for application. We said at the beginning that we would relax the notion of table lookup, allowing an argument to match an input entry if the argument is equal or greater than the input entry. Instead, the `↦-elim` rule seems to require an exact match. However, because of the `sub` rule, application really does allow larger arguments.

```
↦-elim2 : ∀ {Γ} {γ : Env Γ} {M₁ M₂ v₁ v₂ v₃}
  → γ ⊢ M₁ ↓ (v₁ ↦ v₃)
  → γ ⊢ M₂ ↓ v₂
  → v₁ ⊑ v₂
    ------------------
  → γ ⊢ (M₁ · M₂) ↓ v₃
↦-elim2 d₁ d₂ lt = ↦-elim d₁ (sub d₂ lt)
```

Exercise `denot-plus`^{c} (practice)

What is a denotation for `plus`^{c}? That is, find a value `v` (other than `⊥`) such that `∅ ⊢ plus`^{c} `↓ v`. Also, give the proof of `∅ ⊢ plus`^{c} `↓ v` for your choice of `v`.

```
-- Your code goes here
```

Denotations and denotational equality

Next we define a notion of denotational equality based on the above semantics. Its statement makes use of an if-and-only-if, which we define as follows.

```
_iff_ : Set → Set → Set
P iff Q = (P → Q) × (Q → P)
```

Another way to view the denotational semantics is as a function that maps a term to a relation from environments to values. That is, the *denotation* of a term is a relation from environments to values.

```
Denotation : Context → Set₁
Denotation Γ = (Env Γ → Value → Set)
```

The following function $\mathcal{E}$ gives this alternative view of the semantics, which really just amounts to changing the order of the parameters.

```
ℰ : ∀{Γ} → (M : Γ ⊢ ★) → Denotation Γ
ℰ M = λ γ v → γ ⊢ M ↓ v
```

In general, two denotations are equal when they produce the same values in the same environment.

```
infix 3 _≃_

_≃_ : ∀ {Γ} → (Denotation Γ) → (Denotation Γ) → Set
(_≃_ {Γ} D₁ D₂) = (γ : Env Γ) → (v : Value) → D₁ γ v iff D₂ γ v
```

Denotational equality is an equivalence relation.

```
≃-refl : ∀ {Γ : Context} → {M : Denotation Γ}
  → M ≃ M
≃-refl γ v = ⟨ (λ x → x) , (λ x → x) ⟩

≃-sym : ∀ {Γ : Context} → {M N : Denotation Γ}
  → M ≃ N
    -----
  → N ≃ M
≃-sym eq γ v = ⟨ (proj₂ (eq γ v)) , (proj₁ (eq γ v)) ⟩

≃-trans : ∀ {Γ : Context} → {M₁ M₂ M₃ : Denotation Γ}
  → M₁ ≃ M₂
  → M₂ ≃ M₃
    -------
  → M₁ ≃ M₃
≃-trans eq1 eq2 γ v = ⟨ (λ z → proj₁ (eq2 γ v) (proj₁ (eq1 γ v) z)) ,
                        (λ z → proj₂ (eq1 γ v) (proj₂ (eq2 γ v) z)) ⟩
```

Two terms `M` and `N` are denotational equal when their denotations are equal, that is, `ℰ M ≃ ℰ N`.

The following submodule introduces equational reasoning for the `≃` relation.

```
module ≃-Reasoning {Γ : Context} where
  infix 1 start_
  infixr 2 _≃⟨_⟩_ _≃⟨_⟩_
  infix 3 _□

  start_ : ∀ {x y : Denotation Γ}
    → x ≃ y
      -----
    → x ≃ y
  start x≃y = x≃y

  _≃⟨_⟩_ : ∀ (x : Denotation Γ) {y z : Denotation Γ}
    → x ≃ y
    → y ≃ z
      -----
    → x ≃ z
  (x ≃⟨ x≃y ⟩ y≃z) = ≃-trans x≃y y≃z

  _≃⟨⟩_ : ∀ (x : Denotation Γ) {y : Denotation Γ}
    → x ≃ y
      -----
    → x ≃ y
  x ≃⟨⟩ x≃y = x≃y

  _□ : ∀ (x : Denotation Γ)
      -----
    → x ≃ x
  (x □) = ≃-refl
```

Road map for the following chapters

The subsequent chapters prove that the denotational semantics has several desirable properties. First, we prove that the semantics is compositional, i.e., that the denotation of a term is a function of the denotations of its subterms. To do this we shall prove equations of the following shape.

```
ℰ (` x) ≃ ...
ℰ (ƛ M) ≃ ... ℰ M ...
ℰ (M · N) ≃ ... ℰ M ... ℰ N ...
```

The compositionality property is not trivial because the semantics we have defined includes three rules that are not syntax directed: `⊥-intro`, `⊔-intro`, and `sub`. The above equations suggest that the denotational semantics can be defined as a recursive function, and indeed, we give such a definition and prove that it is equivalent to $\mathcal{E}$.

Next we investigate whether the denotational semantics and the reduction semantics are equivalent. Recall that the job of a language semantics is to describe the observable behavior of a given program `M`. For the lambda calculus there are several choices that one can make, but they usually boil down to a single bit of information:

- divergence: the program `M` executes forever.
- termination: the program `M` halts.

We can characterize divergence and termination in terms of reduction.

- divergence: `¬ (M —↠ ƛ N)` for any term `N`.
- termination: `M —↠ ƛ N` for some term `N`.

We can also characterize divergence and termination using denotations.

- divergence: `¬ (∅ ⊢ M ↓ v ↦ w)` for any `v` and `w`.
- termination: `∅ ⊢ M ↓ v ↦ w` for some `v` and `w`.

Alternatively, we can use the denotation function `ℰ`.

- divergence: `¬ (ℰ M ≃ ℰ (ƛ N))` for any term `N`.
- termination: `ℰ M ≃ ℰ (ƛ N)` for some term `N`.

So the question is whether the reduction semantics and denotational semantics are equivalent.

```
(∃ N. M —↠ ƛ N)  iff  (∃ N. ℰ M ≃ ℰ (ƛ N))
```

We address each direction of the equivalence in the second and third chapters. In the second chapter we prove that reduction to a lambda abstraction implies denotational equality to a lambda abstraction. This property is called the *soundness* in the literature.

```
M —↠ ƛ N  implies  ℰ M ≃ ℰ (ƛ N)
```

In the third chapter we prove that denotational equality to a lambda abstraction implies reduction to a lambda abstraction. This property is called *adequacy* in the literature.

```
ℰ M ≃ ℰ (ƛ N)  implies M —↠ ƛ N′ for some N′
```

The fourth chapter applies the results of the three preceding chapters (compositionality, soundness, and adequacy) to prove that denotational equality implies a property called *contextual equivalence*. This property is important because it justifies the use of denotational equality in proving the correctness of program transformations such as performance optimizations.

The proofs of all of these properties rely on some basic results about the denotational semantics, which we establish in the rest of this chapter. We start with some lemmas about renaming, which are quite similar to the renaming lemmas that we have seen in previous chapters. We conclude with a proof of an important inversion lemma for the less-than relation regarding function values.

Renaming preserves denotations

We shall prove that renaming variables, and changing the environment accordingly, preserves the meaning of a term. We generalize the renaming lemma to allow the values in the new environment to be the same or larger than the original values. This generalization is useful in proving that reduction implies denotational equality.

As before, we need an extension lemma to handle the case where we proceed underneath a lambda abstraction. Suppose that `ρ` is a renaming that maps variables in `γ` into variables with equal or larger values in `δ`. This lemmas says that extending the renaming producing a renaming `ext r` that maps `γ , v` to `δ , v`.

```
ext-⊑ : ∀ {Γ Δ v} {γ : Env Γ} {δ : Env Δ}
  → (ρ : Rename Γ Δ)
  → γ `⊑ (δ ∘ ρ)
    ----------------------------
  → (γ `, v) `⊑ ((δ `, v) ∘ ext ρ)
ext-⊑ ρ lt Z = ⊑-refl
ext-⊑ ρ lt (S n′) = lt n′
```

We proceed by cases on the de Bruijn index `n`.

- If it is `Z`, then we just need to show that `v ≡ v`, which we have by `refl`.
- If it is `S n′`, then the goal simplifies to `γ n′ ≡ δ (ρ n′)`, which is an instance of the premise.

Now for the renaming lemma. Suppose we have a renaming that maps variables in `γ` into variables with the same values in `δ`. If `M` results in `v` when evaluated in environment `γ`, then applying the renaming to `M` produces a program that results in the same value `v` when evaluated in `δ`.

```
rename-pres : ∀ {Γ Δ v} {γ : Env Γ} {δ : Env Δ} {M : Γ ⊢ ⋆}
  → (ρ : Rename Γ Δ)
  → γ `⊑ (δ ∘ ρ)
  → γ ⊢ M ↓ v
    ---------------------
  → δ ⊢ (rename ρ M) ↓ v
rename-pres ρ lt (var {x = x}) = sub var (lt x)
rename-pres ρ lt (↦-elim d d₁) =
  ↦-elim (rename-pres ρ lt d) (rename-pres ρ lt d₁)
rename-pres ρ lt (↦-intro d) =
  ↦-intro (rename-pres (ext ρ) (ext-⊑ ρ lt) d)
rename-pres ρ lt ⊥-intro = ⊥-intro
```

```
rename-pres ρ lt (⊔-intro d d₁) =
  ⊔-intro (rename-pres ρ lt d) (rename-pres ρ lt d₁)
rename-pres ρ lt (sub d lt′) =
  sub (rename-pres ρ lt d) lt′
```

The proof is by induction on the semantics of `M`. As you can see, all of the cases are trivial except the cases for variables and lambda.

- For a variable `x`, we make use of the premise to show that `γ x ≡ δ (ρ x)`.
- For a lambda abstraction, the induction hypothesis requires us to extend the renaming. We do so, and use the `ext-⊑` lemma to show that the extended renaming maps variables to ones with equivalent values.

Environment strengthening and identity renaming

We shall need a corollary of the renaming lemma that says that replacing the environment with a larger one (a stronger one) does not change whether a term `M` results in particular value `v`. In particular, if `γ ⊢ M ↓ v` and `γ ⊑ δ`, then `δ ⊢ M ↓ v`. What does this have to do with renaming? It's renaming with the identity function. We apply the renaming lemma with the identity renaming, which gives us `δ ⊢ rename (λ {A} x → x) M ↓ v`, and then we apply the `rename-id` lemma to obtain `δ ⊢ M ↓ v`.

```
⊑-env : ∀ {Γ} {γ : Env Γ} {δ : Env Γ} {M v}
  → γ ⊢ M ↓ v
  → γ `⊑ δ
    ----------
  → δ ⊢ M ↓ v
⊑-env{Γ}{γ}{δ}{M}{v} d lt
      with rename-pres{Γ}{Γ}{v}{γ}{δ}{M} (λ {A} x → x) lt d
... | δ⊢id[M]↓v rewrite rename-id {Γ}{⋆}{M} =
      δ⊢id[M]↓v
```

In the proof that substitution reflects denotations, in the case for lambda abstraction, we use a minor variation of `⊑-env`, in which just the last element of the environment gets larger.

```
up-env : ∀ {Γ} {γ : Env Γ} {M v u₁ u₂}
  → (γ `, u₁) ⊢ M ↓ v
  → u₁ ⊑ u₂
    -----------------
  → (γ `, u₂) ⊢ M ↓ v
up-env d lt = ⊑-env d (ext-le lt)
  where
  ext-le : ∀ {γ u₁ u₂} → u₁ ⊑ u₂ → (γ `, u₁) `⊑ (γ `, u₂)
  ext-le lt Z = lt
  ext-le lt (S n) = ⊑-refl
```

Exercise `denot-church` (recommended)

Church numerals are more general than natural numbers in that they represent paths. A path consists of `n` edges and `n + 1` vertices. We store the vertices in a vector of length `n + 1` in reverse order. The edges in the path map the ith vertex to the `i + 1` vertex. The following

function `D^suc` (for denotation of successor) constructs a table whose entries are all the edges in the path.

```
D^suc : (n : ℕ) → Vec Value (suc n) → Value
D^suc zero (a[0] :: []) = ⊥
D^suc (suc i) (a[i+1] :: a[i] :: ls) = a[i] ↦ a[i+1] ⊔ D^suc i (a[i] :: ls)
```

We use the following auxiliary function to obtain the last element of a non-empty vector. (This formulation is more convenient for our purposes than the one in the Agda standard library.)

```
vec-last : ∀{n : ℕ} → Vec Value (suc n) → Value
vec-last {0} (a :: []) = a
vec-last {suc n} (a :: b :: ls) = vec-last (b :: ls)
```

The function `Dᶜ` computes the denotation of the nth Church numeral for a given path.

```
Dᶜ : (n : ℕ) → Vec Value (suc n) → Value
Dᶜ n (a[n] :: ls) = (D^suc n (a[n] :: ls)) ↦ (vec-last (a[n] :: ls)) ↦ a[n]
```

- The Church numeral for 0 ignores its first argument and returns its second argument, so for the singleton path consisting of just `a[0]`, its denotation is

  ```
  ⊥ ↦ a[0] ↦ a[0]
  ```

- The Church numeral for `suc n` takes two arguments: a successor function whose denotation is given by `D^suc`, and the start of the path (last of the vector). It returns the `n + 1` vertex in the path.

  ```
  (D^suc (suc n) (a[n+1] :: a[n] :: ls)) ↦ (vec-last (a[n] :: ls)) ↦ a[n+1]
  ```

The exercise is to prove that for any path `ls`, the meaning of the Church numeral `n` is `Dᶜ n ls`.

To facilitate talking about arbitrary Church numerals, the following `church` function builds the term for the nth Church numeral, using the auxiliary function `apply-n`.

```
apply-n : (n : ℕ) → ∅ , ★ , ★ ⊢ ★
apply-n zero = # 0
apply-n (suc n) = # 1 · apply-n n

church : (n : ℕ) → ∅ ⊢ ★
church n = ƛ ƛ apply-n n
```

Prove the following theorem.

```
denot-church : ∀{n : ℕ}{ls : Vec Value (suc n)}
   → ` ∅ ⊢ church n ↓ Dᶜ n ls
```

```
-- Your code goes here
```

Inversion of the less-than relation for functions

What can we deduce from knowing that a function `v ↦ w` is less than some value `u`? What can we deduce about `u`? The answer to this question is called the inversion property of less-than for functions. This question is not easy to answer because of the `⊑-dist` rule, which relates a function on the left to a pair of functions on the right. So `u` may include several functions that, as a group, relate to `v ↦ w`. Furthermore, because of the rules `⊑-conj-R1` and `⊑-conj-R2`, there may be other values inside `u`, such as `⊥`, that have nothing to do with `v ↦ w`. But in general, we can deduce that `u` includes a collection of functions where the join of their domains is less than `v` and the join of their codomains is greater than `w`.

To precisely state and prove this inversion property, we need to define what it means for a value to *include* a collection of values. We also need to define how to compute the join of their domains and codomains.

Value membership and inclusion

Recall that we think of a value as a set of entries with the join operator `v ⊔ w` acting like set union. The function value `v ↦ w` and bottom value `⊥` constitute the two kinds of elements of the set. (In other contexts one can instead think of `⊥` as the empty set, but here we must think of it as an element.) We write `u ∈ v` to say that `u` is an element of `v`, as defined below.

```
infix 5 _∈_

_∈_ : Value → Value → Set
u ∈ ⊥ = u ≡ ⊥
u ∈ v ↦ w = u ≡ v ↦ w
u ∈ (v ⊔ w) = u ∈ v ⊎ u ∈ w
```

So we can represent a collection of values simply as a value. We write `v ⊆ w` to say that all the elements of `v` are also in `w`.

```
infix 5 _⊆_

_⊆_ : Value → Value → Set
v ⊆ w = ∀{u} → u ∈ v → u ∈ w
```

The notions of membership and inclusion for values are closely related to the less-than relation. They are narrower relations in that they imply the less-than relation but not the other way around.

```
∈→⊑ : ∀{u v : Value}
    → u ∈ v
      -----
    → u ⊑ v
∈→⊑ {.⊥} {⊥} refl = ⊑-bot
∈→⊑ {v ↦ w} {v ↦ w} refl = ⊑-refl
∈→⊑ {u} {v ⊔ w} (inj₁ x) = ⊑-conj-R1 (∈→⊑ x)
∈→⊑ {u} {v ⊔ w} (inj₂ y) = ⊑-conj-R2 (∈→⊑ y)

⊆→⊑ : ∀{u v : Value}
    → u ⊆ v
      -----
    → u ⊑ v
⊆→⊑ {⊥} s with s {⊥} refl
... | x = ⊑-bot
```

```
⊆→⊑ {u ↦ u′} s with s {u ↦ u′} refl
... | x = ∈→⊑ x
⊆→⊑ {u ⊔ u′} s = ⊑-conj-L (⊆→⊑ (λ z → s (inj₁ z))) (⊆→⊑ (λ z → s (inj₂ z)))
```

We shall also need some inversion principles for value inclusion. If the union of `u` and `v` is included in `w`, then of course both `u` and `v` are each included in `w`.

```
⊔⊆-inv : ∀{u v w : Value}
       → (u ⊔ v) ⊆ w
         ---------------
       → u ⊆ w × v ⊆ w
⊔⊆-inv uvw = ⟨ (λ x → uvw (inj₁ x)) , (λ x → uvw (inj₂ x)) ⟩
```

In our value representation, the function value `v ↦ w` is both an element and also a singleton set. So if `v ↦ w` is a subset of `u`, then `v ↦ w` must be a member of `u`.

```
↦⊆→∈ : ∀{v w u : Value}
     → v ↦ w ⊆ u
       ---------
     → v ↦ w ∈ u
↦⊆→∈ incl = incl refl
```

Function values

To identify collections of functions, we define the following two predicates. We write `Fun u` if `u` is a function value, that is, if `u ≡ v ↦ w` for some values `v` and `w`. We write `all-funs v` if all the elements of `v` are functions.

```
data Fun : Value → Set where
  fun : ∀{u v w} → u ≡ (v ↦ w) → Fun u

all-funs : Value → Set
all-funs v = ∀{u} → u ∈ v → Fun u
```

The value `⊥` is not a function.

```
¬Fun⊥ : ¬ (Fun ⊥)
¬Fun⊥ (fun ())
```

In our values-as-sets representation, our sets always include at least one element. Thus, if all the elements are functions, there is at least one that is a function.

```
all-funs∈ : ∀{u}
  → all-funs u
  → Σ[ v ∈ Value ] Σ[ w ∈ Value ] v ↦ w ∈ u
all-funs∈ {⊥} f with f {⊥} refl
... | fun ()
all-funs∈ {v ↦ w} f = ⟨ v , ⟨ w , refl ⟩ ⟩
all-funs∈ {u ⊔ u′} f
  with all-funs∈ (λ z → f (inj₁ z))
... | ⟨ v , ⟨ w , m ⟩ ⟩ = ⟨ v , ⟨ w , (inj₁ m) ⟩ ⟩
```

Domains and codomains

Returning to our goal, the inversion principle for less-than a function, we want to show that `v ↦ w ⊑ u` implies that `u` includes a set of function values such that the join of their domains is less than `v` and the join of their codomains is greater than `w`.

To this end we define the following `⌊dom` and `⌊cod` functions. Given some value `u` (that represents a set of entries), `⌊dom u` returns the join of their domains and `⌊cod u` returns the join of their codomains.

```
⌊dom : (u : Value) → Value
⌊dom ⊥ = ⊥
⌊dom (v ↦ w) = v
⌊dom (u ⊔ u′) = ⌊dom u ⊔ ⌊dom u′

⌊cod : (u : Value) → Value
⌊cod ⊥ = ⊥
⌊cod (v ↦ w) = w
⌊cod (u ⊔ u′) = ⌊cod u ⊔ ⌊cod u′
```

We need just one property each for `⌊dom` and `⌊cod`. Given a collection of functions represented by value `u`, and an entry `v ↦ w ∈ u`, we know that `v` is included in the domain of `v`.

```
↦∈→⊆⌊dom : ∀{u v w : Value}
           → all-funs u → (v ↦ w) ∈ u
             ----------------------
           → v ⊆ ⌊dom u
↦∈→⊆⌊dom {⊥} fg () u∈v
↦∈→⊆⌊dom {v ↦ w} fg refl u∈v = u∈v
↦∈→⊆⌊dom {u ⊔ u′} fg (inj₁ v↦w∈u) u∈v =
  let ih = ↦∈→⊆⌊dom (λ z → fg (inj₁ z)) v↦w∈u in
  inj₁ (ih u∈v)
↦∈→⊆⌊dom {u ⊔ u′} fg (inj₂ v↦w∈u′) u∈v =
  let ih = ↦∈→⊆⌊dom (λ z → fg (inj₂ z)) v↦w∈u′ in
  inj₂ (ih u∈v)
```

Regarding `⌊cod`, suppose we have a collection of functions represented by `u`, but all of them are just copies of `v ↦ w`. Then the `⌊cod u` is included in `w`.

```
⊆↦→⌊cod⊆ : ∀{u v w : Value}
  → u ⊆ v ↦ w
    ---------
  → ⌊cod u ⊆ w
⊆↦→⌊cod⊆ {⊥} s refl with s {⊥} refl
... | ()
⊆↦→⌊cod⊆ {C ↦ C′} s m with s {C ↦ C′} refl
... | refl = m
⊆↦→⌊cod⊆ {u ⊔ u′} s (inj₁ x) = ⊆↦→⌊cod⊆ (λ {C} z → s (inj₁ z)) x
⊆↦→⌊cod⊆ {u ⊔ u′} s (inj₂ y) = ⊆↦→⌊cod⊆ (λ {C} z → s (inj₂ z)) y
```

With the `⌊dom` and `⌊cod` functions in hand, we can make precise the conclusion of the inversion principle for functions, which we package into the following predicate named `factor`. We say that `v ↦ w` *factors* `u` into `u′` if `u′` is a included in `u`, if `u′` contains only functions, its domain is less than `v`, and its codomain is greater than `w`.

```
factor : (u : Value) → (u′ : Value) → (v : Value) → (w : Value) → Set
factor u u′ v w = all-funs u′ × u′ ⊆ u × v ⌊dom u′⌋ ⊑ v × w ⊑⌊cod u′⌋
```

So the inversion principle for functions can be stated as

```
  v ↦ w ⊑ u
  ---------------
→ factor u u′ v w
```

We prove the inversion principle for functions by induction on the derivation of the less-than relation. To make the induction hypothesis stronger, we broaden the premise `v ↦ w ⊑ u` to `u₁ ⊑ u₂`, and strengthen the conclusion to say that for *every* function value `v ↦ w ∈ u₁`, we have that `v ↦ w` factors `u₂` into some value `u₃`.

```
→ u₁ ⊑ u₂
  ------------------------------------------------------
→ ∀{v w} → v ↦ w ∈ u₁ → Σ[ u₃ ∈ Value ] factor u₂ u₃ v w
```

Inversion of less-than for functions, the case for ⊑-trans

The crux of the proof is the case for `⊑-trans`.

```
u₁ ⊑ u   u ⊑ u₂
--------------- (⊑-trans)
    u₁ ⊑ u₂
```

By the induction hypothesis for `u₁ ⊑ u`, we know that `v ↦ w factors u into u′`, for some value `u′`, so we have `all-funs u′` and `u′ ⊆ u`. By the induction hypothesis for `u ⊑ u₂`, we know that for any `v′ ↦ w′ ∈ u`, `v′ ↦ w′` factors `u₂` into `u₃`. With these facts in hand, we proceed by induction on `u′` to prove that `(⌊dom u′⌋) ↦ (⌊cod u′⌋)` factors `u₂` into `u₃`. We discuss each case of the proof in the text below.

```
sub-inv-trans : ∀{u′ u₂ u : Value}
    → all-funs u′ → u′ ⊆ u
    → (∀{v′ w′} → v′ ↦ w′ ∈ u → Σ[ u₃ ∈ Value ] factor u₂ u₃ v′ w′)
      ------------------------------------------------------------
    → Σ[ u₃ ∈ Value ] factor u₂ u₃ (⌊dom u′⌋) (⌊cod u′⌋)
sub-inv-trans {⊥} {u₂} {u} fu′ u′⊆u IH =
  ⊥-elim (contradiction (fu′ refl) ¬Fun⊥)
sub-inv-trans {u₁′ ↦ u₂′} {u₂} {u} fg u′⊆u IH = IH (↦⊆→∈ u′⊆u)
sub-inv-trans {u₁′ ⊔ u₂′} {u₂} {u} fg u′⊆u IH
    with ⊔⊆-inv u′⊆u
... | ⟨ u₁′⊆u , u₂′⊆u ⟩
    with sub-inv-trans {u₁′} {u₂} {u} (λ {v′} z → fg (inj₁ z)) u₁′⊆u IH
      | sub-inv-trans {u₂′} {u₂} {u} (λ {v′} z → fg (inj₂ z)) u₂′⊆u IH
... | ⟨ u₃₁ , ⟨ fu21' , ⟨ u₃₁⊆u₂ , ⟨ du₃₁⊑du₁′ , cu₁′⊑cu₃₁ ⟩ ⟩ ⟩ ⟩
    | ⟨ u₃₂ , ⟨ fu22' , ⟨ u₃₂⊆u₂ , ⟨ du₃₂⊑du₂′ , cu₁′⊑cu₃₂ ⟩ ⟩ ⟩ ⟩ =
      ⟨ (u₃₁ ⊔ u₃₂) , ⟨ fu₂′ , ⟨ u₂′⊆u₂ ,
      ⟨ ⊔⊑⊔ du₃₁⊑du₁′ du₃₂⊑du₂′ ,
        ⊔⊑⊔ cu₁′⊑cu₃₁ cu₁′⊑cu₃₂ ⟩ ⟩ ⟩ ⟩
    where fu₂′ : {v′ : Value} → v′ ∈ u₃₁ ⊎ v′ ∈ u₃₂ → Fun v′
          fu₂′ {v′} (inj₁ x) = fu21' x
          fu₂′ {v′} (inj₂ y) = fu22' y
          u₂′⊆u₂ : {C : Value} → C ∈ u₃₁ ⊎ C ∈ u₃₂ → C ∈ u₂
          u₂′⊆u₂ {C} (inj₁ x) = u₃₁⊆u₂ x
```

```
        u₂′⊆u₂ {C} (inj₂ y) = u₃₂⊆u₂ y
```

- Suppose `u′ ≡ ⊥`. Then we have a contradiction because it is not the case that `Fun ⊥`.
- Suppose `u′ ≡ u₁′ ↦ u₂′`. Then `u₁′ ↦ u₂′ ∈ u` and we can apply the premise (the induction hypothesis from `u ⊑ u₂`) to obtain that `u₁′ ↦ u₂′` factors of `u₂ into u₂′`. This case is complete because `⌊dom u′ ≡ u₁′` and `⌊cod u′ ≡ u₂′`.
- Suppose `u′ ≡ u₁′ ⊔ u₂′`. Then we have `u₁′ ⊆ u` and `u₂′ ⊆ u`. We also have `all-funs u₁′` and `all-funs u₂′`, so we can apply the induction hypothesis for both `u₁′` and `u₂′`. So there exists values `u₃₁` and `u₃₂` such that `(⌊dom u₁′) ↦ (⌊cod u₁′)` factors `u` into `u₃₁` and `(⌊dom u₂′) ↦ (⌊cod u₂′)` factors `u` into `u₃₂`. We will show that `(⌊dom u) ↦ (⌊cod u)` factors `u` into `u₃₁ ⊔ u₃₂`. So we need to show that

  ```
  ⌊dom (u₃₁ ⊔ u₃₂) ⊑ ⌊dom (u₁′ ⊔ u₂′)
  ⌊cod (u₁′ ⊔ u₂′) ⊑ ⌊cod (u₃₁ ⊔ u₃₂)
  ```

  But those both follow directly from the factoring of `u` into `u₃₁` and `u₃₂`, using the monotonicity of `⊔` with respect to `⊑`.

Inversion of less-than for functions

We come to the proof of the main lemma concerning the inversion of less-than for functions. We show that if `u₁ ⊑ u₂`, then for any `v ↦ w ∈ u₁`, we can factor `u₂` into `u₃` according to `v ↦ w`. We proceed by induction on the derivation of `u₁ ⊑ u₂`, and describe each case in the text after the Agda proof.

```
sub-inv : ∀{u₁ u₂ : Value}
        → u₁ ⊑ u₂
        → ∀{v w} → v ↦ w ∈ u₁
          ------------------------------------
        → Σ[ u₃ ∈ Value ] factor u₂ u₃ v w
sub-inv {⊥} {u₂} ⊑-bot {v} {w} ()
sub-inv {u₁₁ ⊔ u₁₂} {u₂} (⊑-conj-L lt1 lt2) {v} {w} (inj₁ x) = sub-inv lt1 x
sub-inv {u₁₁ ⊔ u₁₂} {u₂} (⊑-conj-L lt1 lt2) {v} {w} (inj₂ y) = sub-inv lt2 y
sub-inv {u₁} {u₂₁ ⊔ u₂₂} (⊑-conj-R1 lt) {v} {w} m
    with sub-inv lt m
... | ⟨ u₃₁ , ⟨ fu₃₁ , ⟨ u₃₁⊆u₂₁ , ⟨ domu₃₁⊑v , w⊑codu₃₁ ⟩ ⟩ ⟩ ⟩ =
      ⟨ u₃₁ , ⟨ fu₃₁ , ⟨ (λ {w} z → inj₁ (u₃₁⊆u₂₁ z)) ,
                                ⟨ domu₃₁⊑v , w⊑codu₃₁ ⟩ ⟩ ⟩ ⟩
sub-inv {u₁} {u₂₁ ⊔ u₂₂} (⊑-conj-R2 lt) {v} {w} m
    with sub-inv lt m
... | ⟨ u₃₂ , ⟨ fu₃₂ , ⟨ u₃₂⊆u₂₂ , ⟨ domu₃₂⊑v , w⊑codu₃₂ ⟩ ⟩ ⟩ ⟩ =
      ⟨ u₃₂ , ⟨ fu₃₂ , ⟨ (λ {C} z → inj₂ (u₃₂⊆u₂₂ z)) ,
                                ⟨ domu₃₂⊑v , w⊑codu₃₂ ⟩ ⟩ ⟩ ⟩
sub-inv {u₁} {u₂} (⊑-trans{v = u} u₁⊑u u⊑u₂) {v} {w} v↦w∈u₁
    with sub-inv u₁⊑u v↦w∈u₁
... | ⟨ u′ , ⟨ fu′ , ⟨ u′⊆u , ⟨ domu′⊑v , w⊑codu′ ⟩ ⟩ ⟩ ⟩
    with sub-inv-trans {u′} fu′ u′⊆u (sub-inv u⊑u₂)
... | ⟨ u₃ , ⟨ fu₃ , ⟨ u₃⊆u₂ , ⟨ domu₃⊑domu′ , codu′⊑codu₃ ⟩ ⟩ ⟩ ⟩ =
      ⟨ u₃ , ⟨ fu₃ , ⟨ u₃⊆u₂ , ⟨ ⊑-trans domu₃⊑domu′ domu′⊑v ,
                               ⊑-trans w⊑codu′ codu′⊑codu₃ ⟩ ⟩ ⟩ ⟩
sub-inv {u₁₁ ↦ u₁₂} {u₂₁ ↦ u₂₂} (⊑-fun lt1 lt2) refl =
    ⟨ u₂₁ ↦ u₂₂ , ⟨ (λ {w} → fun) , ⟨ (λ {C} z → z) , ⟨ lt1 , lt2 ⟩ ⟩ ⟩ ⟩
```

```
sub-inv {u₂₁ ↦ (u₂₂ ⊔ u₂₃)} {u₂₁ ↦ u₂₂ ⊔ u₂₁ ↦ u₂₃} ⊑-dist
    {.u₂₁} {.(u₂₂ ⊔ u₂₃)} refl =
    ⟨ u₂₁ ↦ u₂₂ ⊔ u₂₁ ↦ u₂₃ , ⟨ f , ⟨ g , ⟨ ⊑-conj-L ⊑-refl ⊑-refl , ⊑-refl ⟩ ⟩ ⟩ ⟩
  where f : all-funs (u₂₁ ↦ u₂₂ ⊔ u₂₁ ↦ u₂₃)
        f (inj₁ x) = fun x
        f (inj₂ y) = fun y
        g : (u₂₁ ↦ u₂₂ ⊔ u₂₁ ↦ u₂₃) ⊆ (u₂₁ ↦ u₂₂ ⊔ u₂₁ ↦ u₂₃)
        g (inj₁ x) = inj₁ x
        g (inj₂ y) = inj₂ y
```

Let `v` and `w` be arbitrary values.

- Case `⊑-bot`. So `u₁ ≡ ⊥`. We have `v ↦ w ∈ ⊥`, but that is impossible.
- Case `⊑-conj-L`.

  ```
  u₁₁ ⊑ u₂   u₁₂ ⊑ u₂
  -------------------
  u₁₁ ⊔ u₁₂ ⊑ u₂
  ```

  Given that `v ↦ w ∈ u₁₁ ⊔ u₁₂`, there are two subcases to consider.

  - Subcase `v ↦ w ∈ u₁₁`. We conclude by the induction hypothesis for `u₁₁ ⊑ u₂`.
  - Subcase `v ↦ w ∈ u₁₂`. We conclude by the induction hypothesis for `u₁₂ ⊑ u₂`.

- Case `⊑-conj-R1`.

  ```
  u₁ ⊑ u₂₁
  --------------
  u₁ ⊑ u₂₁ ⊔ u₂₂
  ```

  Given that `v ↦ w ∈ u₁`, the induction hypothesis for `u₁ ⊑ u₂₁` gives us that `v ↦ w` factors `u₂₁` into `u₃₁` for some `u₃₁`. To show that `v ↦ w` also factors `u₂₁ ⊔ u₂₂` into `u₃₁`, we just need to show that `u₃₁ ⊆ u₂₁ ⊔ u₂₂`, but that follows directly from `u₃₁ ⊆ u₂₁`.

- Case `⊑-conj-R2`. This case follows by reasoning similar to the case for `⊑-conj-R1`.
- Case `⊑-trans`.

  ```
  u₁ ⊑ u   u ⊑ u₂
  ---------------
      u₁ ⊑ u₂
  ```

  By the induction hypothesis for `u₁ ⊑ u`, we know that `v ↦ w` factors `u` into `u′`, for some value `u′`, so we have `all-funs u′` and `u′ ⊆ u`. By the induction hypothesis for `u ⊑ u₂`, we know that for any `v′ ↦ w′ ∈ u`, `v′ ↦ w′` factors `u₂`. Now we apply the lemma sub-inv-trans, which gives us some `u₃` such that `(⌊dom u′) ↦ (⌊cod u′)` factors `u₂` into `u₃`. We show that `v ↦ w` also factors `u₂` into `u₃`. From `⌊dom u₃ ⊑ ⌊dom u′` and `⌊dom u′ ⊑ v`, we have `⌊dom u₃ ⊑ v`. From `w ⊑ ⌊cod u′` and `⌊cod u′ ⊑ ⌊cod u₃`, we have `w ⊑ ⌊cod u₃`, and this case is complete.

- Case `⊑-fun`.

  ```
  u₂₁ ⊑ u₁₁  u₁₂ ⊑ u₂₂
  --------------------
  u₁₁ ↦ u₁₂ ⊑ u₂₁ ↦ u₂₂
  ```

Given that `v ↦ w ∈ u₁₁ ↦ u₁₂`, we have `v ≡ u₁₁` and `w ≡ u₁₂`. We show that `u₁₁ ↦ u₁₂` factors `u₂₁ ↦ u₂₂` into itself. We need to show that `⌊dom (u₂₁ ↦ u₂₂) ⊑ u₁₁` and `u₁₂ ⊑ ⌊cod (u₂₁ ↦ u₂₂)`, but that is equivalent to our premises `u₂₁ ⊑ u₁₁` and `u₁₂ ⊑ u₂₂`.

- Case `⊑-dist`.

```
      ------------------------------------------
      u₂₁ ↦ (u₂₂ ⊔ u₂₃) ⊑ (u₂₁ ↦ u₂₂) ⊔ (u₂₁ ↦ u₂₃)
```

Given that `v ↦ w ∈ u₂₁ ↦ (u₂₂ ⊔ u₂₃)`, we have `v ≡ u₂₁` and `w ≡ u₂₂ ⊔ u₂₃`. We show that `u₂₁ ↦ (u₂₂ ⊔ u₂₃)` factors `(u₂₁ ↦ u₂₂) ⊔ (u₂₁ ↦ u₂₃)` into itself. We have `u₂₁ ⊔ u₂₁ ⊑ u₂₁`, and also `u₂₂ ⊔ u₂₃ ⊑ u₂₂ ⊔ u₂₃`, so the proof is complete.

We conclude this section with two corollaries of the sub-inv lemma. First, we have the following property that is convenient to use in later proofs. We specialize the premise to just `v ↦ w ⊑ u₁` and we modify the conclusion to say that for every `v′ ↦ w′ ∈ u₂`, we have `v′ ⊑ v`.

```
sub-inv-fun : ∀{v w u₁ : Value}
  → (v ↦ w) ⊑ u₁
      ------------------------------------------------------
  → Σ[ u₂ ∈ Value ] all-funs u₂ × u₂ ⊆ u₁
      × (∀{v′ w′} → (v′ ↦ w′) ∈ u₂ → v′ ⊑ v) × w ⊑ ⌊cod u₂
sub-inv-fun{v}{w}{u₁} abc
  with sub-inv abc {v}{w} refl
... | ⟨ u₂ , ⟨ f , ⟨ u₂⊆u₁ , ⟨ db , cc ⟩ ⟩ ⟩ ⟩ =
      ⟨ u₂ , ⟨ f , ⟨ u₂⊆u₁ , ⟨ G , cc ⟩ ⟩ ⟩ ⟩
  where G : ∀{D E} → (D ↦ E) ∈ u₂ → D ⊑ v
        G{D}{E} m = ⊑-trans (⊆→⊑ (↦∈→⊆⌊dom f m)) db
```

The second corollary is the inversion rule that one would expect for less-than with functions on the left and right-hand sides.

```
↦⊑↦-inv : ∀{v w v′ w′}
        → v ↦ w ⊑ v′ ↦ w′
          -----------------
        → v′ ⊑ v × w ⊑ w′
↦⊑↦-inv{v}{w}{v′}{w′} lt
    with sub-inv-fun lt
... | ⟨ Γ , ⟨ f , ⟨ Γ⊆v34 , ⟨ lt1 , lt2 ⟩ ⟩ ⟩ ⟩
    with all-funs∈ f
... | ⟨ u , ⟨ u′ , u↦u′∈Γ ⟩ ⟩
    with Γ⊆v34 u↦u′∈Γ
... | refl =
  let ⌊cod⌋Γ⊆w′ = ⊆↦↦⌊cod⌋⊆ Γ⊆v34 in
  ⟨ lt1 u↦u′∈Γ , ⊑-trans lt2 (⊆→⊑ ⌊cod⌋Γ⊆w′) ⟩
```

Notes

The denotational semantics presented in this chapter is an example of a *filter model* (Barendregt, Coppo, Dezani-Ciancaglini, 1983). Filter models use type systems with intersection types to precisely characterize runtime behavior (Coppo, Dezani-Ciancaglini, and Salle, 1979). The notation that we use in this chapter is not that of type systems and intersection types, but the `Value` data type is isomorphic to types (`↦` is `→`, `⊔` is `∧`, `⊥` is `⊤`), the `⊑` relation is the inverse of subtyping

`<:`, and the evaluation relation `ρ ⊢ M ↓ v` is isomorphic to a type system. Write `Γ` instead of `ρ`, `A` instead of `v`, and replace `↓` with `:` and one has a typing judgement `Γ ⊢ M : A`. By varying the definition of subtyping and using different choices of type atoms, intersection type systems provide semantics for many different untyped λ calculi, from full beta to the lazy and call-by-value calculi (Alessi, Barbanera, and Dezani-Ciancaglini, 2006) (Rocca and Paolini, 2004). The denotational semantics in this chapter corresponds to the BCD system (Barendregt, Coppo, Dezani-Ciancaglini, 1983). Part 3 of the book *Lambda Calculus with Types* describes a framework for intersection type systems that enables results similar to the ones in this chapter, but for the entire family of intersection type systems (Barendregt, Dekkers, and Statman, 2013).

The two ideas of using finite tables to represent functions and of relaxing table lookup to enable self application first appeared in a technical report by Gordon Plotkin (1972) and are later described in an article in Theoretical Computer Science (Plotkin 1993). In that work, the inductive definition of `Value` is a bit different than the one we use:

```
Value = C + ℘f(Value) × ℘f(Value)
```

where `C` is a set of constants and `℘f` means finite powerset. The pairs in `℘f(Value) × ℘f(Value)` represent input-output mappings, just as in this chapter. The finite powersets are used to enable a function table to appear in the input and in the output. These differences amount to changing where the recursion appears in the definition of `Value`. Plotkin's model is an example of a *graph model* of the untyped lambda calculus (Barendregt, 1984). In a graph model, the semantics is presented as a function from programs and environments to (possibly infinite) sets of values. The semantics in this chapter is instead defined as a relation, but set-valued functions are isomorphic to relations. Indeed, we present the semantics as a function in the next chapter and prove that it is equivalent to the relational version.

Dana Scott's ℘(ω) (1976) and Engeler's B(A) (1981) are two more examples of graph models. Both use the following inductive definition of `Value`.

```
Value = C + ℘f(Value) × Value
```

The use of `Value` instead of `℘f(Value)` in the output does not restrict expressiveness compared to Plotkin's model because the semantics use sets of values and a pair of sets `(V, V′)` can be represented as a set of pairs `{ (V, v′) | v′ ∈ V′ }`. In Scott's ℘(ω), the above values are mapped to and from the natural numbers using a kind of Godel encoding.

References

- Intersection Types and Lambda Models. Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini, Theoretical Compututer Science, vol. 355, pages 108-126, 2006.
- The Lambda Calculus. H.P. Barendregt, 1984.
- A filter lambda model and the completeness of type assignment. Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini, Journal of Symbolic Logic, vol. 48, pages 931-940, 1983.
- Lambda Calculus with Types. Henk Barendregt, Wil Dekkers, and Richard Statman, Cambridge University Press, Perspectives in Logic,

2013.

- Functional characterization of some semantic equalities inside λ-calculus. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Salle, in Sixth Colloquium on Automata, Languages and Programming. Springer, pages 133–146, 1979.

- Algebras and combinators. Erwin Engeler, Algebra Universalis, vol. 13, pages 389-392, 1981.
- A Set-Theoretical Definition of Application. Gordon D. Plotkin, University of Edinburgh, Technical Report MIP-R-95, 1972.
- Set-theoretical and other elementary models of the λ-calculus. Gordon D. Plotkin, Theoretical Computer Science, vol. 121, pages 351-409, 1993.
- The Parametric Lambda Calculus. Simona Ronchi Della Rocca and Luca Paolini, Springer, 2004.
- Data Types as Lattices. Dana Scott, SIAM Journal on Computing, vol. 5, pages 522-587, 1976.

Unicode

This chapter uses the following unicode:

```
⊥  U+22A5  UP TACK (\bot)
↦  U+21A6  RIGHTWARDS ARROW FROM BAR (\mapsto)
⊔  U+2294  SQUARE CUP (\lub)
⊑  U+2291  SQUARE IMAGE OF OR EQUAL TO (\sqsubseteq)
⨆  U+2A06  N-ARY SQUARE UNION OPERATOR (\Lub)
⊢  U+22A2  RIGHT TACK (\|- or \vdash)
↓  U+2193  DOWNWARDS ARROW (\d)
ᶜ  U+1D9C  MODIFIER LETTER SMALL C (\^c)
ℰ  U+2130  SCRIPT CAPITAL E (\McE)
≃  U+2243  ASYMPTOTICALLY EQUAL TO (\~- or \simeq)
∈  U+2208  ELEMENT OF (\in)
⊆  U+2286  SUBSET OF OR EQUAL TO (\sub= or \subseteq)
```

Chapter 21

Compositional: The denotational semantics is compositional

```
module plfa.part3.Compositional where
```

Introduction

In this chapter we prove that the denotational semantics is compositional, which means we fill in the ellipses in the following equations.

```
ℰ (` x) ≃ ...
ℰ (ƛ M) ≃ ... ℰ M ...
ℰ (M · N) ≃ ... ℰ M ... ℰ N ...
```

Such equations would imply that the denotational semantics could be instead defined as a recursive function. Indeed, we end this chapter with such a definition and prove that it is equivalent to $\mathcal{E}$.

Imports

```
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Data.Unit using (⊤; tt)
open import plfa.part2.Untyped
  using (Context; _,_; ★; _∋_; _⊢_; `_; ƛ_; _·_)
open import plfa.part3.Denotational
  using (Value; _↦_; _`,_; _⊔_; ⊥; _⊑_; _⊢_↓_;
         ⊑-bot; ⊑-fun; ⊑-conj-L; ⊑-conj-R1; ⊑-conj-R2;
         ⊑-dist; ⊑-refl; ⊑-trans; ⊔↦⊔-dist;
         var; ↦-intro; ↦-elim; ⊔-intro; ⊥-intro; sub;
         up-env; ℰ; _≃_; ≃-sym; Denotation; Env)
open plfa.part3.Denotational.≃-Reasoning
```

Equation for lambda abstraction

Regarding the first equation

```
ℰ (ƛ M) ≃ ... ℰ M ...
```

we need to define a function that maps a `Denotation (Γ , ⋆)` to a `Denotation Γ`. This function, let us name it `ℱ`, should mimic the non-recursive part of the semantics when applied to a lambda term. In particular, we need to consider the rules `↦-intro`, `⊥-intro`, and `⊔-intro`. So `ℱ` has three parameters, the denotation `D` of the subterm `M`, an environment `γ`, and a value `v`. If we define `ℱ` by recursion on the value `v`, then it matches up nicely with the three rules `↦-intro`, `⊥-intro`, and `⊔-intro`.

```
ℱ : ∀{Γ} → Denotation (Γ , ⋆) → Denotation Γ
ℱ D γ (v ↦ w) = D (γ `, v) w
ℱ D γ ⊥ = ⊤
ℱ D γ (u ⊔ v) = (ℱ D γ u) × (ℱ D γ v)
```

If one squints hard enough, the `ℱ` function starts to look like the `curry` operation familiar to functional programmers. It turns a function that expects a tuple of length `n + 1` (the environment `Γ , ⋆`) into a function that expects a tuple of length `n` and returns a function of one parameter.

Using this `ℱ`, we hope to prove that

```
ℰ (ƛ N) ≃ ℱ (ℰ N)
```

The function `ℱ` is preserved when going from a larger value `v` to a smaller value `u`. The proof is a straightforward induction on the derivation of `u ⊑ v`, using the `up-env` lemma in the case for the `⊑-fun` rule.

```
sub-ℱ : ∀{Γ}{N : Γ , ⋆ ⊢ ⋆}{γ v u}
  → ℱ (ℰ N) γ v
  → u ⊑ v
    ------------
  → ℱ (ℰ N) γ u
sub-ℱ d ⊑-bot = tt
sub-ℱ d (⊑-fun lt lt′) = sub (up-env d lt) lt′
sub-ℱ d (⊑-conj-L lt lt₁) = ⟨ sub-ℱ d lt , sub-ℱ d lt₁ ⟩
sub-ℱ d (⊑-conj-R1 lt) = sub-ℱ (proj₁ d) lt
sub-ℱ d (⊑-conj-R2 lt) = sub-ℱ (proj₂ d) lt
sub-ℱ {v = v₁ ↦ v₂ ⊔ v₁ ↦ v₃} {v₁ ↦ (v₂ ⊔ v₃)} ⟨ N2 , N3 ⟩ ⊑-dist =
  ⊔-intro N2 N3
sub-ℱ d (⊑-trans x₁ x₂) = sub-ℱ (sub-ℱ d x₂) x₁
```

With this subsumption property in hand, we can prove the forward direction of the semantic equation for lambda. The proof is by induction on the semantics, using `sub-ℱ` in the case for the `sub` rule.

```
ℰƛ→ℱℰ : ∀{Γ}{γ : Env Γ}{N : Γ , ⋆ ⊢ ⋆}{v : Value}
  → ℰ (ƛ N) γ v
    ------------
  → ℱ (ℰ N) γ v
ℰƛ→ℱℰ (↦-intro d) = d
ℰƛ→ℱℰ ⊥-intro = tt
```

```
ℰ↛ℱ (⊔-intro d₁ d₂) = ⟨ ℰ↛ℱ d₁ , ℰ↛ℱ d₂ ⟩
ℰ↛ℱ (sub d lt) = sub-ℱ (ℰ↛ℱ d) lt
```

The "inversion lemma" for lambda abstraction is a special case of the above. The inversion lemma is useful in proving that denotations are preserved by reduction.

```
lambda-inversion : ∀{Γ}{γ : Env Γ}{N : Γ , ⋆ ⊢ ⋆}{v₁ v₂ : Value}
  → γ ⊢ ƛ N ↓ v₁ ↦ v₂
    -----------------
  → (γ `, v₁) ⊢ N ↓ v₂
lambda-inversion{v₁ = v₁}{v₂ = v₂} d = ℰ↛ℱ{v = v₁ ↦ v₂} d
```

The backward direction of the semantic equation for lambda is even easier to prove than the forward direction. We proceed by induction on the value v.

```
ℱ↛ℰ : ∀{Γ}{γ : Env Γ}{N : Γ , ⋆ ⊢ ⋆}{v : Value}
  → ℱ (ℰ N) γ v
    ------------
  → ℰ (ƛ N) γ v
ℱ↛ℰ {v = ⊥} d = ⊥-intro
ℱ↛ℰ {v = v₁ ↦ v₂} d = ↦-intro d
ℱ↛ℰ {v = v₁ ⊔ v₂} ⟨ d1 , d2 ⟩ = ⊔-intro (ℱ↛ℰ d1) (ℱ↛ℰ d2)
```

So indeed, the denotational semantics is compositional with respect to lambda abstraction, as witnessed by the function `ℱ`.

```
lam-equiv : ∀{Γ}{N : Γ , ⋆ ⊢ ⋆}
  → ℰ (ƛ N) ≃ ℱ (ℰ N)
lam-equiv γ v = ⟨ ℰ↛ℱ , ℱ↛ℰ ⟩
```

Equation for function application

Next we fill in the ellipses for the equation concerning function application.

```
ℰ (M · N) ≃ ... ℰ M ... ℰ N ...
```

For this we need to define a function that takes two denotations, both in context `Γ`, and produces another one in context `Γ`. This function, let us name it `●`, needs to mimic the non-recursive aspects of the semantics of an application `L · M`. We cannot proceed as easily as for `ℱ` and define the function by recursion on value `v` because, for example, the rule `↦-elim` applies to any value. Instead we shall define `●` in a way that directly deals with the `↦-elim` and `⊥-intro` rules but ignores `⊔-intro`. This makes the forward direction of the proof more difficult, and the case for `⊔-intro` demonstrates why the `⊑-dist` rule is important.

So we define the application of `D₁` to `D₂`, written `D₁ ● D₂`, to include any value `w` equivalent to `⊥`, for the `⊥-intro` rule, and to include any value `w` that is the output of an entry `v ↦ w` in `D₁`, provided the input `v` is in `D₂`, for the `↦-elim` rule.

```
infixl 7 _●_

_●_ : ∀{Γ} → Denotation Γ → Denotation Γ → Denotation Γ
(D₁ ● D₂) γ w = w ⊑ ⊥ ⊎ Σ[ v ∈ Value ]( D₁ γ (v ↦ w) × D₂ γ v )
```

If one squints hard enough, the `_●_` operator starts to look like the `apply` operation familiar to functional programmers. It takes two parameters and applies the first to the second.

Next we consider the inversion lemma for application, which is also the forward direction of the semantic equation for application. We describe the proof below.

```
𝒞·→●𝒞 : ∀{Γ}{γ : Env Γ}{L M : Γ ⊢ ⋆}{v : Value}
  → 𝒞 (L · M) γ v
    ----------------
  → (𝒞 L ● 𝒞 M) γ v
𝒞·→●𝒞 (↦-elim{v = v′} d₁ d₂) = inj₂ ⟨ v′ , ⟨ d₁ , d₂ ⟩ ⟩
𝒞·→●𝒞 {v = ⊥} ⊥-intro = inj₁ ⊑-bot
𝒞·→●𝒞 {Γ}{γ}{L}{M}{v} (⊔-intro{v = v₁}{w = v₂} d₁ d₂)
    with 𝒞·→●𝒞 d₁ | 𝒞·→●𝒞 d₂
... | inj₁ lt1 | inj₁ lt2 = inj₁ (⊑-conj-L lt1 lt2)
... | inj₁ lt1 | inj₂ ⟨ v₁′ , ⟨ L↓v12 , M↓v3 ⟩ ⟩ =
      inj₂ ⟨ v₁′ , ⟨ sub L↓v12 lt , M↓v3 ⟩ ⟩
      where lt : v₁′ ↦ (v₁ ⊔ v₂) ⊑ v₁′ ↦ v₂
            lt = (⊑-fun ⊑-refl (⊑-conj-L (⊑-trans lt1 ⊑-bot) ⊑-refl))
... | inj₂ ⟨ v₁′ , ⟨ L↓v12 , M↓v3 ⟩ ⟩ | inj₁ lt2 =
      inj₂ ⟨ v₁′ , ⟨ sub L↓v12 lt , M↓v3 ⟩ ⟩
      where lt : v₁′ ↦ (v₁ ⊔ v₂) ⊑ v₁′ ↦ v₁
            lt = (⊑-fun ⊑-refl (⊑-conj-L ⊑-refl (⊑-trans lt2 ⊑-bot)))
... | inj₂ ⟨ v₁′ , ⟨ L↓v12 , M↓v3 ⟩ ⟩ | inj₂ ⟨ v₁′′ , ⟨ L↓v12′ , M↓v3′ ⟩ ⟩ =
      let L↓⊔ = ⊔-intro L↓v12 L↓v12′ in
      let M↓⊔ = ⊔-intro M↓v3 M↓v3′ in
      inj₂ ⟨ v₁′ ⊔ v₁′′ , ⟨ sub L↓⊔ ⊔↦⊔-dist , M↓⊔ ⟩ ⟩
𝒞·→●𝒞 {Γ}{γ}{L}{M}{v} (sub d lt)
    with 𝒞·→●𝒞 d
... | inj₁ lt2 = inj₁ (⊑-trans lt lt2)
... | inj₂ ⟨ v₁ , ⟨ L↓v12 , M↓v3 ⟩ ⟩ =
      inj₂ ⟨ v₁ , ⟨ sub L↓v12 (⊑-fun ⊑-refl lt) , M↓v3 ⟩ ⟩
```

We proceed by induction on the semantics.

- In case `↦-elim` we have `γ ⊢ L ↓ (v′ ↦ v)` and `γ ⊢ M ↓ v′`, which is all we need to show `(𝒞 L ● 𝒞 M) γ v`.
- In case `⊥-intro` we have `v = ⊥`. We conclude that `v ⊑ ⊥`.
- In case `⊔-intro` we have `𝒞 (L · M) γ v₁` and `𝒞 (L · M) γ v₂` and need to show `(𝒞 L ● 𝒞 M) γ (v₁ ⊔ v₂)`. By the induction hypothesis, we have `(𝒞 L ● 𝒞 M) γ v₁` and `(𝒞 L ● 𝒞 M) γ v₂`. We have four subcases to consider.
  - Suppose `v₁ ⊑ ⊥` and `v₂ ⊑ ⊥`. Then `v₁ ⊔ v₂ ⊑ ⊥`.
  - Suppose `v₁ ⊑ ⊥`, `γ ⊢ L ↓ v₁′ ↦ v₂`, and `γ ⊢ M ↓ v₁′`. We have `γ ⊢ L ↓ v₁′ ↦ (v₁ ⊔ v₂)` by rule `sub` because `v₁′ ↦ (v₁ ⊔ v₂) ⊑ v₁′ ↦ v₂`.
  - Suppose `γ ⊢ L ↓ v₁′ ↦ v₁`, `γ ⊢ M ↓ v₁′`, and `v₂ ⊑ ⊥`. We have `γ ⊢ L ↓ v₁′ ↦ (v₁ ⊔ v₂)` by rule `sub` because `v₁′ ↦ (v₁ ⊔ v₂) ⊑ v₁′ ↦ v₁`.
  - Suppose `γ ⊢ L ↓ v₁′′ ↦ v₁`, `γ ⊢ M ↓ v₁′′`, `γ ⊢ L ↓ v₁′ ↦ v₂`, and `γ ⊢ M ↓ v₁′`. This case is the most interesting. By two uses of the rule `⊔-intro` we have `γ ⊢ L ↓ (v₁′ ↦ v₂) ⊔ (v₁′′ ↦ v₁)` and `γ ⊢ M ↓ (v₁′ ⊔ v₁′′)`. But this does not yet match what we need for `𝒞 L ● 𝒞 M` because the result of `L` must be an `↦` whose input entry is `v₁′ ⊔ v₁′′`. So we use the `sub` rule to obtain

`γ ⊢ L ↓ (v₁′ ⊔ v₁′′) ↦ (v₁ ⊔ v₂)`, using the `⊔↦⊔-dist` lemma (thanks to the `⊑-dist` rule) to show that

```
(v₁′ ⊔ v₁′′) ↦ (v₁ ⊔ v₂) ⊑ (v₁′ ↦ v₂) ⊔ (v₁′′ ↦ v₁)
```

So we have proved what is needed for this case.

- In case `sub` we have `Γ ⊢ L · M ↓ v₁` and `v ⊑ v₁`. By the induction hypothesis, we have `(ℰ L ● ℰ M) γ v₁`. We have two subcases to consider.
  - Suppose `v₁ ⊑ ⊥`. We conclude that `v ⊑ ⊥`.
  - Suppose `Γ ⊢ L ↓ v′ → v₁` and `Γ ⊢ M ↓ v′`. We conclude with `Γ ⊢ L ↓ v′ → v` by rule `sub`, because `v′ → v ⊑ v′ → v₁`.

The forward direction is proved by cases on the premise `(ℰ L ● ℰ M) γ v`. In case `v ⊑ ⊥`, we obtain `Γ ⊢ L · M ↓ ⊥` by rule `⊥-intro`. Otherwise, we conclude immediately by rule `↦-elim`.

```
●ℰ→ℰ· : ∀{Γ}{γ : Env Γ}{L M : Γ ⊢ ★}{v}
  → (ℰ L ● ℰ M) γ v
    ----------------
  → ℰ (L · M) γ v
●ℰ→ℰ· {γ}{v} (inj₁ lt) = sub ⊥-intro lt
●ℰ→ℰ· {γ}{v} (inj₂ ⟨ v₁ , ⟨ d1 , d2 ⟩ ⟩) = ↦-elim d1 d2
```

So we have proved that the semantics is compositional with respect to function application, as witnessed by the `●` function.

```
app-equiv : ∀{Γ}{L M : Γ ⊢ ★}
  → ℰ (L · M) ≃ (ℰ L) ● (ℰ M)
app-equiv γ v = ⟨ ℰ·→●ℰ , ●ℰ→ℰ· ⟩
```

We also need an inversion lemma for variables. If `Γ ⊢ x ↓ v`, then `v ⊑ γ x`. The proof is a straightforward induction on the semantics.

```
var-inv : ∀ {Γ v x} {γ : Env Γ}
  → ℰ (` x) γ v
    ------------------
  → v ⊑ γ x
var-inv (var) = ⊑-refl
var-inv (⊔-intro d₁ d₂) = ⊑-conj-L (var-inv d₁) (var-inv d₂)
var-inv (sub d lt) = ⊑-trans lt (var-inv d)
var-inv ⊥-intro = ⊑-bot
```

To round-out the semantic equations, we establish the following one for variables.

```
var-equiv : ∀{Γ}{x : Γ ∋ ★} → ℰ (` x) ≃ (λ γ v → v ⊑ γ x)
var-equiv γ v = ⟨ var-inv , (λ lt → sub var lt) ⟩
```

Congruence

The main work of this chapter is complete: we have established semantic equations that show how the denotational semantics is compositional. In this section and the next we make use of these equations to prove some corollaries: that denotational equality is a *congruence* and to prove

the *compositionality property*, which states that surrounding two denotationally-equal terms in the same context produces two programs that are denotationally equal.

We begin by showing that denotational equality is a congruence with respect to lambda abstraction: that `ℰ N ≃ ℰ N′` implies `ℰ (ƛ N) ≃ ℰ (ƛ N′)`. We shall use the `lam-equiv` equation to reduce this question to whether `ℱ` is a congruence.

```
ℱ-cong : ∀{Γ}{D D′ : Denotation (Γ , ⋆)}
  → D ≃ D′
    -----------
  → ℱ D ≃ ℱ D′
ℱ-cong{Γ} D≃D′ γ v =
  ⟨ (λ x → ℱ≃{γ}{v} x D≃D′) , (λ x → ℱ≃{γ}{v} x (≃-sym D≃D′)) ⟩
  where
  ℱ≃ : ∀{γ : Env Γ}{v}{D D′ : Denotation (Γ , ⋆)}
    → ℱ D γ v → D ≃ D′ → ℱ D′ γ v
  ℱ≃ {v = ⊥} fd dd′ = tt
  ℱ≃ {γ}{v ↦ w} fd dd′ = proj₁ (dd′ (γ `, v) w) fd
  ℱ≃ {γ}{u ⊔ w} fd dd′ = ⟨ ℱ≃{γ}{u} (proj₁ fd) dd′ , ℱ≃{γ}{w} (proj₂ fd) dd′ ⟩
```

The proof of `ℱ-cong` uses the lemma `ℱ≃` to handle both directions of the if-and-only-if. That lemma is proved by a straightforward induction on the value `v`.

We now prove that lambda abstraction is a congruence by direct equational reasoning.

```
lam-cong : ∀{Γ}{N N′ : Γ , ⋆ ⊢ ⋆}
  → ℰ N ≃ ℰ N′
    -----------------
  → ℰ (ƛ N) ≃ ℰ (ƛ N′)
lam-cong {Γ}{N}{N′} N≃N′ =
  start
    ℰ (ƛ N)
  ≃⟨ lam-equiv ⟩
    ℱ (ℰ N)
  ≃⟨ ℱ-cong N≃N′ ⟩
    ℱ (ℰ N′)
  ≃⟨ ≃-sym lam-equiv ⟩
    ℰ (ƛ N′)
  □
```

Next we prove that denotational equality is a congruence for application: that `ℰ L ≃ ℰ L′` and `ℰ M ≃ ℰ M′` imply `ℰ (L · M) ≃ ℰ (L′ · M′)`. The `app-equiv` equation reduces this to the question of whether the `●` operator is a congruence.

```
●-cong : ∀{Γ}{D₁ D₁′ D₂ D₂′ : Denotation Γ}
  → D₁ ≃ D₁′ → D₂ ≃ D₂′
  → (D₁ ● D₂) ≃ (D₁′ ● D₂′)
●-cong {Γ} d1 d2 γ v = ⟨ (λ x → ●≃ x d1 d2) ,
                         (λ x → ●≃ x (≃-sym d1) (≃-sym d2)) ⟩
  where
  ●≃ : ∀{γ : Env Γ}{v}{D₁ D₁′ D₂ D₂′ : Denotation Γ}
    → (D₁ ● D₂) γ v → D₁ ≃ D₁′ → D₂ ≃ D₂′
    → (D₁′ ● D₂′) γ v
  ●≃ (inj₁ v⊑⊥) eq₁ eq₂ = inj₁ v⊑⊥
  ●≃ {γ} {w} (inj₂ ⟨ v , ⟨ Dv↦w , Dv ⟩ ⟩) eq₁ eq₂ =
    inj₂ ⟨ v , ⟨ proj₁ (eq₁ γ (v ↦ w)) Dv↦w , proj₁ (eq₂ γ v) Dv ⟩ ⟩
```

Again, both directions of the if-and-only-if are proved via a lemma. This time the lemma is proved

by cases on `(D₁ ● D₂) γ v`.

With the congruence of `●`, we can prove that application is a congruence by direct equational reasoning.

```
app-cong : ∀{Γ}{L L′ M M′ : Γ ⊢ ★}
  → ℰ L ≃ ℰ L′
  → ℰ M ≃ ℰ M′
    ------------------------
  → ℰ (L · M) ≃ ℰ (L′ · M′)
app-cong {Γ}{L}{L′}{M}{M′} L≅L′ M≅M′ =
  start
    ℰ (L · M)
  ≃⟨ app-equiv ⟩
    ℰ L ● ℰ M
  ≃⟨ ●-cong L≅L′ M≅M′ ⟩
    ℰ L′ ● ℰ M′
  ≃⟨ ≃-sym app-equiv ⟩
    ℰ (L′ · M′)
  □
```

Compositionality

The *compositionality property* states that surrounding two terms that are denotationally equal in the same context produces two programs that are denotationally equal. To make this precise, we define what we mean by “context” and “surround”.

A *context* is a program with one hole in it. The following data definition `Ctx` makes this idea explicit. We index the `Ctx` data type with two contexts for variables: one for the hole and one for terms that result from filling the hole.

```
data Ctx : Context → Context → Set where
  ctx-hole : ∀{Γ} → Ctx Γ Γ
  ctx-lam : ∀{Γ Δ} → Ctx (Γ , ★) (Δ , ★) → Ctx (Γ , ★) Δ
  ctx-app-L : ∀{Γ Δ} → Ctx Γ Δ → Δ ⊢ ★ → Ctx Γ Δ
  ctx-app-R : ∀{Γ Δ} → Δ ⊢ ★ → Ctx Γ Δ → Ctx Γ Δ
```

- The constructor `ctx-hole` represents the hole, and in this case the variable context for the hole is the same as the variable context for the term that results from filling the hole.
- The constructor `ctx-lam` takes a `Ctx` and produces a larger one that adds a lambda abstraction at the top. The variable context of the hole stays the same, whereas we remove one variable from the context of the resulting term because it is bound by this lambda abstraction.
- There are two constructions for application, `ctx-app-L` and `ctx-app-R`. The `ctx-app-L` is for when the hole is inside the left-hand term (the operator) and the later is when the hole is inside the right-hand term (the operand).

The action of surrounding a term with a context is defined by the following `plug` function. It is defined by recursion on the context.

```
plug : ∀{Γ}{Δ} → Ctx Γ Δ → Γ ⊢ ★ → Δ ⊢ ★
plug ctx-hole M = M
```

```
plug (ctx-lam C) N = ƛ plug C N
plug (ctx-app-L C N) L = (plug C L) · N
plug (ctx-app-R L C) M = L · (plug C M)
```

We are ready to state and prove the compositionality principle. Given two terms `M` and `N` that are denotationally equal, plugging them both into an arbitrary context `C` produces two programs that are denotationally equal.

```
compositionality : ∀{Γ Δ}{C : Ctx Γ Δ} {M N : Γ ⊢ ⋆}
  → ℰ M ≃ ℰ N
    ---------------------------
  → ℰ (plug C M) ≃ ℰ (plug C N)
compositionality {C = ctx-hole} M≃N =
  M≃N
compositionality {C = ctx-lam C′} M≃N =
  lam-cong (compositionality {C = C′} M≃N)
compositionality {C = ctx-app-L C′ L} M≃N =
  app-cong (compositionality {C = C′} M≃N) λ γ v → ⟨ (λ x → x) , (λ x → x) ⟩
compositionality {C = ctx-app-R L C′} M≃N =
  app-cong (λ γ v → ⟨ (λ x → x) , (λ x → x) ⟩) (compositionality {C = C′} M≃N)
```

The proof is a straightforward induction on the context `C`, using the congruence properties `lam-cong` and `app-cong` that we established above.

The denotational semantics defined as a function

Having established the three equations `var-equiv`, `lam-equiv`, and `app-equiv`, one should be able to define the denotational semantics as a recursive function over the input term `M`. Indeed, we define the following function `⟦ M ⟧` that maps terms to denotations, using the auxiliary curry `𝒯` and apply `●` functions in the cases for lambda and application, respectively.

```
⟦_⟧ : ∀{Γ} → (M : Γ ⊢ ⋆) → Denotation Γ
⟦ ` x ⟧ γ v = v ⊑ γ x
⟦ ƛ N ⟧ = 𝒯 ⟦ N ⟧
⟦ L · M ⟧ = ⟦ L ⟧ ● ⟦ M ⟧
```

The proof that `ℰ M` is denotationally equal to `⟦ M ⟧` is a straightforward induction, using the three equations `var-equiv`, `lam-equiv`, and `app-equiv` together with the congruence lemmas for `𝒯` and `●`.

```
ℰ≃⟦⟧ : ∀ {Γ} {M : Γ ⊢ ⋆} → ℰ M ≃ ⟦ M ⟧
ℰ≃⟦⟧ {Γ} {` x} = var-equiv
ℰ≃⟦⟧ {Γ} {ƛ N} =
  let ih = ℰ≃⟦⟧ {M = N} in
    ℰ (ƛ N)
  ≃⟨ lam-equiv ⟩
    𝒯 (ℰ N)
  ≃⟨ 𝒯-cong (ℰ≃⟦⟧ {M = N}) ⟩
    𝒯 ⟦ N ⟧
  ≃⟨⟩
    ⟦ ƛ N ⟧
  □
```

```
ℰ≃⟦⟧ {Γ} {L · M} =
    ℰ (L · M)
  ≃⟨ app-equiv ⟩
    ℰ L ● ℰ M
  ≃⟨ ●-cong (ℰ≃⟦⟧ {M = L}) (ℰ≃⟦⟧ {M = M}) ⟩
    ⟦ L ⟧ ● ⟦ M ⟧
  ≃⟨⟩
    ⟦ L · M ⟧
  □
```

Unicode

This chapter uses the following unicode:

```
ℱ  U+2131  SCRIPT CAPITAL F (\McF)
●  U+2131  BLACK CIRCLE (\cib)
```

Chapter 22

Soundness: Soundness of reduction with respect to denotational semantics

```
module plfa.part3.Soundness where
```

Introduction

In this chapter we prove that the reduction semantics is sound with respect to the denotational semantics, i.e., for any term L

```
L —↠ ƛ N  implies  ℰ L ≃ ℰ (ƛ N)
```

The proof is by induction on the reduction sequence, so the main lemma concerns a single reduction step. We prove that if any term `M` steps to a term `N`, then `M` and `N` are denotationally equal. We shall prove each direction of this if-and-only-if separately. One direction will look just like a type preservation proof. The other direction is like proving type preservation for reduction going in reverse. Recall that type preservation is sometimes called subject reduction. Preservation in reverse is a well-known property and is called *subject expansion*. It is also well-known that subject expansion is false for most typed lambda calculi!

Imports

```
open import Relation.Binary.PropositionalEquality
  using (_≡_; _≢_; refl; sym; cong; cong₂; cong-app)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Agda.Primitive using (lzero)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Negation using (contradiction)
open import Data.Empty using (⊥-elim)
open import Relation.Nullary using (Dec; yes; no)
open import Function using (_∘_)
```

```
open import plfa.part2.Untyped
    using (Context; _,_; _∋_; _⊢_; ★; Z; S_; `_; ƛ_; _·_;
           subst; _[_]; subst-zero; ext; rename; exts;
           _—→_; ξ₁; ξ₂; β; ζ; _—↠_; _—→⟨_⟩_; _∎)
open import plfa.part2.Substitution using (Rename; Subst; ids)
open import plfa.part3.Denotational
    using (Value; ⊥; Env; _⊢_↓_; _`,_; _⊑_; _`⊑_; `⊥; _`⊔_; init; last; init-last;
           ⊑-refl; ⊑-trans; `⊑-refl; ⊑-env; ⊑-env-conj-R1; ⊑-env-conj-R2; up-env;
           var; ↦-elim; ↦-intro; ⊥-intro; ⊔-intro; sub;
           rename-pres; 𝒞; _≃_; ≃-trans)
open import plfa.part3.Compositional using (lambda-inversion; var-inv)
```

Forward reduction preserves denotations

The proof of preservation in this section mixes techniques from previous chapters. Like the proof of preservation for the STLC, we are preserving a relation defined separately from the syntax, in contrast to the intrinsically-typed terms. On the other hand, we are using de Bruijn indices for variables.

The outline of the proof remains the same in that we must prove lemmas concerning all of the auxiliary functions used in the reduction relation: substitution, renaming, and extension.

Simultaneous substitution preserves denotations

Our next goal is to prove that simultaneous substitution preserves meaning. That is, if `M` results in `v` in environment `γ`, then applying a substitution `σ` to `M` gives us a program that also results in `v`, but in an environment `δ` in which, for every variable `x`, `σ x` results in the same value as the one for `x` in the original environment `γ`. We write `δ ⊢ σ ↓ γ` for this condition.

```
infix 3 _`⊢_↓_
_`⊢_↓_ : ∀{Δ Γ} → Env Δ → Subst Γ Δ → Env Γ → Set
_`⊢_↓_ {Δ}{Γ} δ σ γ = (∀ (x : Γ ∋ ★) → δ ⊢ σ x ↓ γ x)
```

As usual, to prepare for lambda abstraction, we prove an extension lemma. It says that applying the `exts` function to a substitution produces a new substitution that maps variables to terms that when evaluated in `δ , v` produce the values in `γ , v`.

```
subst-ext : ∀ {Γ Δ v} {γ : Env Γ} {δ : Env Δ}
  → (σ : Subst Γ Δ)
  → δ `⊢ σ ↓ γ
    -------------------------
  → δ `, v `⊢ exts σ ↓ γ `, v
subst-ext σ d Z = var
subst-ext σ d (S x′) = rename-pres S_ (λ _ → ⊑-refl) (d x′)
```

The proof is by cases on the de Bruijn index `x`.

- If it is `Z`, then we need to show that `δ , v ⊢ # 0 ↓ v`, which we have by rule `var`.
- If it is `S x′`,then we need to show that `δ , v ⊢ rename S_ (σ x′) ↓ γ x′`, which we obtain by the `rename-pres` lemma.

With the extension lemma in hand, the proof that simultaneous substitution preserves meaning is straightforward. Let's dive in!

```
subst-pres : ∀ {Γ Δ v} {γ : Env Γ} {δ : Env Δ} {M : Γ ⊢ ⋆}
  → (σ : Subst Γ Δ)
  → δ ⊢ σ ↓ γ
  → γ ⊢ M ↓ v
    ------------------
  → δ ⊢ subst σ M ↓ v
subst-pres σ s (var {x = x}) = (s x)
subst-pres σ s (↦-elim d₁ d₂) =
  ↦-elim (subst-pres σ s d₁) (subst-pres σ s d₂)
subst-pres σ s (↦-intro d) =
  ↦-intro (subst-pres (λ {A} → exts σ) (subst-ext σ s) d)
subst-pres σ s ⊥-intro = ⊥-intro
subst-pres σ s (⊔-intro d₁ d₂) =
  ⊔-intro (subst-pres σ s d₁) (subst-pres σ s d₂)
subst-pres σ s (sub d lt) = sub (subst-pres σ s d) lt
```

The proof is by induction on the semantics of `M`. The two interesting cases are for variables and lambda abstractions.

- For a variable `x`, we have that `v ⊑ γ x` and we need to show that `δ ⊢ σ x ↓ v`. From the premise applied to `x`, we have that `δ ⊢ σ x ↓ γ x`, so we conclude by the `sub` rule.
- For a lambda abstraction, we must extend the substitution for the induction hypothesis. We apply the `subst-ext` lemma to show that the extended substitution maps variables to terms that result in the appropriate values.

Single substitution preserves denotations

For β reduction, `(ƛ N) · M ⟶ N [ M ]`, we need to show that the semantics is preserved when substituting `M` for de Bruijn index 0 in term `N`. By inversion on the rules `↦-elim` and `↦-intro`, we have that `γ , v ⊢ M ↓ w` and `γ ⊢ N ↓ v`. So we need to show that `γ ⊢ M [ N ] ↓ w`, or equivalently, that `γ ⊢ subst (subst-zero N) M ↓ w`.

```
substitution : ∀ {Γ} {γ : Env Γ} {N M v w}
  → γ `, v ⊢ N ↓ w
  → γ ⊢ M ↓ v
    ---------------
  → γ ⊢ N [ M ] ↓ w
substitution{Γ}{γ}{N}{M}{v}{w} dn dm =
  subst-pres (subst-zero M) sub-z-ok dn
  where
  sub-z-ok : γ `⊢ subst-zero M ↓ (γ `, v)
  sub-z-ok Z = dm
  sub-z-ok (S x) = var
```

This result is a corollary of the lemma for simultaneous substitution. To use the lemma, we just need to show that `subst-zero M` maps variables to terms that produces the same values as those in `γ , v`. Let `y` be an arbitrary variable (de Bruijn index).

- If it is `Z`, then `(subst-zero M) y = M` and `(γ , v) y = v`. By the premise we conclude that `γ ⊢ M ↓ v`.

- If it is `S x`, then `(subst-zero M) (S x) = x` and `(γ , v) (S x) = γ x`. So we conclude that `γ ⊢ x ↓ γ x` by rule `var`.

Reduction preserves denotations

With the substitution lemma in hand, it is straightforward to prove that reduction preserves denotations.

```
preserve : ∀ {Γ} {γ : Env Γ} {M N v}
  → γ ⊢ M ↓ v
  → M ⟶ N
    ----------
  → γ ⊢ N ↓ v
preserve (var) ()
preserve (↦-elim d₁ d₂) (ξ₁ r) = ↦-elim (preserve d₁ r) d₂
preserve (↦-elim d₁ d₂) (ξ₂ r) = ↦-elim d₁ (preserve d₂ r)
preserve (↦-elim d₁ d₂) β = substitution (lambda-inversion d₁) d₂
preserve (↦-intro d) (ζ r) = ↦-intro (preserve d r)
preserve ⊥-intro r = ⊥-intro
preserve (⊔-intro d d₁) r = ⊔-intro (preserve d r) (preserve d₁ r)
preserve (sub d lt) r = sub (preserve d r) lt
```

We proceed by induction on the semantics of `M` with case analysis on the reduction.

- If `M` is a variable, then there is no such reduction.
- If `M` is an application, then the reduction is either a congruence (ξ₁ or ξ₂) or β. For each congruence, we use the induction hypothesis. For β reduction we use the substitution lemma and the `sub` rule.
- The rest of the cases are straightforward.

Reduction reflects denotations

This section proves that reduction reflects the denotation of a term. That is, if `N` results in `v`, and if `M` reduces to `N`, then `M` also results in `v`. While there are some broad similarities between this proof and the above proof of semantic preservation, we shall require a few more technical lemmas to obtain this result.

The main challenge is dealing with the substitution in β reduction:

```
(ƛ N) · M ⟶ N [ M ]
```

We have that `γ ⊢ N [ M ] ↓ v` and need to show that `γ ⊢ (ƛ N) · M ↓ v`. Now consider the derivation of `γ ⊢ N [ M ] ↓ v`. The term `M` may occur 0, 1, or many times inside `N [ M ]`. At each of those occurrences, `M` may result in a different value. But to build a derivation for `(ƛ N) · M`, we need a single value for `M`. If `M` occurred more than 1 time, then we can join all of the different values using `⊔`. If `M` occurred 0 times, then we do not need any information about `M` and can therefore use `⊥` for the value of `M`.

Renaming reflects meaning

Previously we showed that renaming variables preserves meaning. Now we prove the opposite, that it reflects meaning. That is, if `δ ⊢ rename ρ M ↓ v`, then `γ ⊢ M ↓ v`, where `(δ ∘ ρ) ⊑ γ′`.

First, we need a variant of a lemma given earlier.

```
ext-⊑′ : ∀ {Γ Δ v} {γ : Env Γ} {δ : Env Δ}
  → (ρ : Rename Γ Δ)
  → (δ ∘ ρ) `⊑ γ
    ------------------------------
  → ((δ `, v) ∘ ext ρ) `⊑ (γ `, v)
ext-⊑′ ρ lt Z = ⊑-refl
ext-⊑′ ρ lt (S x) = lt x
```

The proof is then as follows.

```
rename-reflect : ∀ {Γ Δ v} {γ : Env Γ} {δ : Env Δ} { M : Γ ⊢ ★}
  → {ρ : Rename Γ Δ}
  → (δ ∘ ρ) `⊑ γ
  → δ ⊢ rename ρ M ↓ v
    ------------------------------------
  → γ ⊢ M ↓ v
rename-reflect {M = ` x} all-n d with var-inv d
... | lt = sub var (⊑-trans lt (all-n x))
rename-reflect {M = ƛ N}{ρ = ρ} all-n (↦-intro d) =
  ↦-intro (rename-reflect (ext-⊑′ ρ all-n) d)
rename-reflect {M = ƛ N} all-n ⊥-intro = ⊥-intro
rename-reflect {M = ƛ N} all-n (⊔-intro d₁ d₂) =
  ⊔-intro (rename-reflect all-n d₁) (rename-reflect all-n d₂)
rename-reflect {M = ƛ N} all-n (sub d₁ lt) =
  sub (rename-reflect all-n d₁) lt
rename-reflect {M = L · M} all-n (↦-elim d₁ d₂) =
  ↦-elim (rename-reflect all-n d₁) (rename-reflect all-n d₂)
rename-reflect {M = L · M} all-n ⊥-intro = ⊥-intro
rename-reflect {M = L · M} all-n (⊔-intro d₁ d₂) =
  ⊔-intro (rename-reflect all-n d₁) (rename-reflect all-n d₂)
rename-reflect {M = L · M} all-n (sub d₁ lt) =
  sub (rename-reflect all-n d₁) lt
```

We cannot prove this lemma by induction on the derivation of `δ ⊢ rename ρ M ↓ v`, so instead we proceed by induction on `M`.

- If it is a variable, we apply the inversion lemma to obtain that `v ⊑ δ (ρ x)`. Instantiating the premise to `x` we have `δ (ρ x) = γ x`, so we conclude by the `var` rule.
- If it is a lambda abstraction `ƛ N`, we have rename `ρ (ƛ N) = ƛ (rename (ext ρ) N)`. We proceed by cases on `δ ⊢ ƛ (rename (ext ρ) N) ↓ v`.
  - Rule `↦-intro`: To satisfy the premise of the induction hypothesis, we prove that the renaming can be extended to be a mapping from `γ , v₁` to `δ , v₁`.
  - Rule `⊥-intro`: We simply apply `⊥-intro`.
  - Rule `⊔-intro`: We apply the induction hypotheses and `⊔-intro`.
  - Rule `sub`: We apply the induction hypothesis and `sub`.

- If it is an application `L · M`, we have `rename ρ (L · M) = (rename ρ L) · (rename ρ M)`. We proceed by cases on `δ ⊢ (rename ρ L) · (rename ρ M) ↓ v` and all the cases are straightforward.

In the upcoming uses of `rename-reflect`, the renaming will always be the increment function. So we prove a corollary for that special case.

```
rename-inc-reflect : ∀ {Γ v′ v} {γ : Env Γ} { M : Γ ⊢ ⋆}
  → (γ `, v′) ⊢ rename S_ M ↓ v
    ----------------------------
  → γ ⊢ M ↓ v
rename-inc-reflect d = rename-reflect `⊑-refl d
```

Substitution reflects denotations, the variable case

We are almost ready to begin proving that simultaneous substitution reflects denotations. That is, if `γ ⊢ (subst σ M) ↓ v`, then `γ ⊢ σ k ↓ δ k` and `δ ⊢ M ↓ v` for any `k` and some `δ`. We shall start with the case in which `M` is a variable `x`. So instead the premise is `γ ⊢ σ x ↓ v` and we need to show that `δ ⊢ x ↓ v` for some `δ`. The `δ` that we choose shall be the environment that maps `x` to `v` and every other variable to `⊥`.

Next we define the environment that maps `x` to `v` and every other variable to `⊥`, that is `const-env x v`. To tell variables apart, we define the following function for deciding equality of variables.

```
_var≟_ : ∀ {Γ} → (x y : Γ ∋ ⋆) → Dec (x ≡ y)
Z var≟ Z = yes refl
Z var≟ (S _) = no λ()
(S _) var≟ Z = no λ()
(S x) var≟ (S y) with x var≟ y
... |                yes refl = yes refl
... |                no neq =   no λ{refl → neq refl}

var≟-refl : ∀ {Γ} (x : Γ ∋ ⋆) → (x var≟ x) ≡ yes refl
var≟-refl Z = refl
var≟-refl (S x) rewrite var≟-refl x = refl
```

Now we use `var≟` to define `const-env`.

```
const-env : ∀{Γ} → (x : Γ ∋ ⋆) → Value → Env Γ
const-env x v y with x var≟ y
... | yes _ = v
... | no _  = ⊥
```

Of course, `const-env x v` maps `x` to value `v`

```
same-const-env : ∀{Γ} {x : Γ ∋ ⋆} {v} → (const-env x v) x ≡ v
same-const-env {x = x} rewrite var≟-refl x = refl
```

and `const-env x v` maps `y` to `⊥, so long as` `x ≢ y`.

```
diff-const-env : ∀{Γ} {x y : Γ ∋ ⋆} {v}
  → x ≢ y
    ------------------
  → const-env x v y ≡ ⊥
diff-const-env {Γ} {x} {y} neq with x var≟ y
... | yes eq = ⊥-elim (neq eq)
... | no _   = refl
```

So we choose `const-env x v` for `δ` and obtain `δ ⊢ x ↓ v` with the `var` rule.

It remains to prove that `γ ⊢ σ ↓ δ` and `δ ⊢ M ↓ v` for any `k`, given that we have chosen `const-env x v` for `δ`. We shall have two cases to consider, `x ≡ y` or `x ≢ y`.

Now to finish the two cases of the proof.

- In the case where `x ≡ y`, we need to show that `γ ⊢ σ y ↓ v`, but that's just our premise.
- In the case where `x ≢ y,` we need to show that `γ ⊢ σ y ↓ ⊥`, which we do via rule `⊥-intro`.

Thus, we have completed the variable case of the proof that simultaneous substitution reflects denotations. Here is the proof again, formally.

```
subst-reflect-var : ∀ {Γ Δ} {γ : Env Δ} {x : Γ ∋ ⋆} {v} {σ : Subst Γ Δ}
  → γ ⊢ σ x ↓ v
    ---------------------------------------
  → Σ[ δ ∈ Env Γ ] γ ⊢ σ ↓ δ × δ ⊢ ` x ↓ v
subst-reflect-var {Γ}{Δ}{γ}{x}{v}{σ} xv
  rewrite sym (same-const-env {Γ}{x}{v}) =
    ⟨ const-env x v , ⟨ const-env-ok , var ⟩ ⟩
  where
  const-env-ok : γ ⊢ σ ↓ const-env x v
  const-env-ok y with x var≟ y
  ... | yes x≡y rewrite sym x≡y | same-const-env {Γ}{x}{v} = xv
  ... | no x≢y rewrite diff-const-env {Γ}{x}{y}{v} x≢y = ⊥-intro
```

Substitutions and environment construction

Every substitution produces terms that can evaluate to `⊥`.

```
subst-⊥ : ∀{Γ Δ}{γ : Env Δ}{σ : Subst Γ Δ}
  -----------------
  → γ ⊢ σ ↓ ⊥
subst-⊥ x = ⊥-intro
```

If a substitution produces terms that evaluate to the values in both `γ₁` and `γ₂`, then those terms also evaluate to the values in `γ₁ ⊔ γ₂`.

```
subst-⊔ : ∀{Γ Δ}{γ : Env Δ}{γ₁ γ₂ : Env Γ}{σ : Subst Γ Δ}
          → γ ⊢ σ ↓ γ₁
          → γ ⊢ σ ↓ γ₂
            ------------------------
          → γ ⊢ σ ↓ (γ₁ ⊔ γ₂)
subst-⊔ γ₁-ok γ₂-ok x = ⊔-intro (γ₁-ok x) (γ₂-ok x)
```

The Lambda constructor is injective

```
lambda-inj : ∀ {Γ} {M N : Γ , ⋆ ⊢ ⋆ }
  → _≡_ {A = Γ ⊢ ⋆} (ƛ M) (ƛ N)
    --------------------------
  → M ≡ N
lambda-inj refl = refl
```

Simultaneous substitution reflects denotations

In this section we prove a central lemma, that substitution reflects denotations. That is, if `γ ⊢ subst σ M ↓ v`, then `δ ⊢ M ↓ v` and `γ ⊢ σ ↓ δ` for some `δ`. We shall proceed by induction on the derivation of `γ ⊢ subst σ M ↓ v`. This requires a minor restatement of the lemma, changing the premise to `γ ⊢ L ↓ v` and `L ≡ subst σ M`.

```
split : ∀ {Γ} {M : Γ , ⋆ ⊢ ⋆} {δ : Env (Γ , ⋆)} {v}
  → δ ⊢ M ↓ v
    -------------------------
  → (init δ `, last δ) ⊢ M ↓ v
split {δ = δ} δMv rewrite init-last δ = δMv

subst-reflect : ∀ {Γ Δ} {δ : Env Δ} {M : Γ ⊢ ⋆} {v} {L : Δ ⊢ ⋆} {σ : Subst Γ Δ}
  → δ ⊢ L ↓ v
  → subst σ M ≡ L
    ------------------------------------
  → Σ[ γ ∈ Env Γ ] δ `⊢ σ ↓ γ × γ ⊢ M ↓ v

subst-reflect {M = M}{σ = σ} (var {x = y}) eqL with M
... | ` x with var {x = y}
... | yv           rewrite sym eqL = subst-reflect-var {σ = σ} yv
subst-reflect {M = M} (var {x = y}) () | M₁ · M₂
subst-reflect {M = M} (var {x = y}) () | ƛ M′

subst-reflect {M = M}{σ = σ} (↦-elim d₁ d₂) eqL
  with M
... | ` x with ↦-elim d₁ d₂
... | d′ rewrite sym eqL = subst-reflect-var {σ = σ} d′
subst-reflect (↦-elim d₁ d₂) () | ƛ M′
subst-reflect{Γ}{Δ}{γ}{σ = σ} (↦-elim d₁ d₂)
  refl | M₁ · M₂
      with subst-reflect {M = M₁} d₁ refl | subst-reflect {M = M₂} d₂ refl
... | ⟨ δ₁ , ⟨ subst-δ₁ , m1 ⟩ ⟩ | ⟨ δ₂ , ⟨ subst-δ₂ , m2 ⟩ ⟩ =
      ⟨ δ₁ `⊔ δ₂ , ⟨ subst-⊔ {γ₁ = δ₁}{γ₂ = δ₂}{σ = σ} subst-δ₁ subst-δ₂ ,
                    ↦-elim (⊑-env m1 (⊑-env-conj-R1 δ₁ δ₂))
                           (⊑-env m2 (⊑-env-conj-R2 δ₁ δ₂)) ⟩ ⟩

subst-reflect {M = M}{σ = σ} (↦-intro d) eqL with M
... | ` x with (↦-intro d)
... | d′ rewrite sym eqL = subst-reflect-var {σ = σ} d′
subst-reflect {σ = σ} (↦-intro d) eq | ƛ M′
  with subst-reflect {σ = exts σ} d (lambda-inj eq)
... | ⟨ δ′ , ⟨ exts-σ-δ′ , m′ ⟩ ⟩ =
    ⟨ init δ′ , ⟨ ((λ x → rename-inc-reflect (exts-σ-δ′ (S x)))) ,
            ↦-intro (up-env (split m′) (var-inv (exts-σ-δ′ Z))) ⟩ ⟩
subst-reflect (↦-intro d) () | M₁ · M₂
```

```
subst-reflect {σ = σ} ⊥-intro eq =
    ⟨ `⊥ , ⟨ subst-⊥ {σ = σ} , ⊥-intro ⟩ ⟩

subst-reflect {σ = σ} (⊔-intro d₁ d₂) eq
  with subst-reflect {σ = σ} d₁ eq | subst-reflect {σ = σ} d₂ eq
... | ⟨ δ₁ , ⟨ subst-δ₁ , m1 ⟩ ⟩ | ⟨ δ₂ , ⟨ subst-δ₂ , m2 ⟩ ⟩ =
      ⟨ δ₁ `⊔ δ₂ , ⟨ subst-⊔ {γ₁ = δ₁}{γ₂ = δ₂}{σ = σ} subst-δ₁ subst-δ₂ ,
                    ⊔-intro (⊑-env m1 (⊑-env-conj-R1 δ₁ δ₂))
                            (⊑-env m2 (⊑-env-conj-R2 δ₁ δ₂)) ⟩ ⟩
subst-reflect (sub d lt) eq
    with subst-reflect d eq
... | ⟨ δ , ⟨ subst-δ , m ⟩ ⟩ = ⟨ δ , ⟨ subst-δ , sub m lt ⟩ ⟩
```

- Case `var`: We have subst `σ M ≡ y`, so `M` must also be a variable, say `x`. We apply the lemma `subst-reflect-var` to conclude.
- Case `↦-elim`: We have `subst σ M ≡ L₁ · L₂`. We proceed by cases on `M`.
  - Case `M ≡ x`: We apply the `subst-reflect-var` lemma again to conclude.
  - Case `M ≡ M₁ · M₂`: By the induction hypothesis, we have some `δ₁` and `δ₂` such that `δ₁ ⊢ M₁ ↓ v₁ ↦ v₃` and `γ ⊢ σ ↓ δ₁`, as well as `δ₂ ⊢ M₂ ↓ v₁` and `γ ⊢ σ ↓ δ₂`. By `⊑-env` we have `δ₁ ⊔ δ₂ ⊢ M₁ ↓ v₁ ↦ v₃` and `δ₁ ⊔ δ₂ ⊢ M₂ ↓ v₁` (using `⊑-env-conj-R1` and `⊑-env-conj-R2`), and therefore `δ₁ ⊔ δ₂ ⊢ M₁ · M₂ ↓ v₃`. We conclude this case by obtaining `γ ⊢ σ ↓ δ₁ ⊔ δ₂` by the `subst-⊔` lemma.
- Case `↦-intro`: We have `subst σ M ≡ ƛ L′`. We proceed by cases on `M`.
  - Case `M ≡ x`: We apply the `subst-reflect-var` lemma.
  - Case `M ≡ ƛ M′`: By the induction hypothesis, we have `(δ′ , v′) ⊢ M′ ↓ v₂` and `(δ , v₁) ⊢ exts σ ↓ (δ′ , v′)`. From the later we have `(δ , v₁) ⊢ # 0 ↓ v′`. By the lemma `var-inv` we have `v′ ⊑ v₁`, so by the `up-env` lemma we have `(δ′ , v₁) ⊢ M′ ↓ v₂` and therefore `δ′ ⊢ ƛ M′ ↓ v₁ → v₂`. We also need to show that `δ ⊢ σ ↓ δ′`. Fix `k`. We have `(δ , v₁) ⊢ rename S_ σ k ↓ δ k′`. We then apply the lemma `rename-inc-reflect` to obtain `δ ⊢ σ k ↓ δ k′`, so this case is complete.
- Case `⊥-intro`: We choose `⊥` for `δ`. We have `⊥ ⊢ M ↓ ⊥` by `⊥-intro`. We have `δ ⊢ σ ↓ ⊥` by the lemma `subst-empty`.
- Case `⊔-intro`: By the induction hypothesis we have `δ₁ ⊢ M ↓ v₁`, `δ₂ ⊢ M ↓ v₂`, `δ ⊢ σ ↓ δ₁`, and `δ ⊢ σ ↓ δ₂`. We have `δ₁ ⊔ δ₂ ⊢ M ↓ v₁` and `δ₁ ⊔ δ₂ ⊢ M ↓ v₂` by `⊑-env` with `⊑-env-conj-R1` and `⊑-env-conj-R2`. So by `⊔-intro` we have `δ₁ ⊔ δ₂ ⊢ M ↓ v₁ ⊔ v₂`. By `subst-⊔` we conclude that `δ ⊢ σ ↓ δ₁ ⊔ δ₂`.

Single substitution reflects denotations

Most of the work is now behind us. We have proved that simultaneous substitution reflects denotations. Of course, β reduction uses single substitution, so we need a corollary that proves that single substitution reflects denotations. That is, given terms `N : (Γ , ⋆ ⊢ ⋆)` and `M : (Γ ⊢ ⋆)`, if `γ ⊢ N [ M ] ↓ w`, then `γ ⊢ M ↓ v` and `(γ , v) ⊢ N ↓ w` for some value `v`. We have `N [ M ] = subst (subst-zero M) N`.

We first prove a lemma about `subst-zero`, that if `δ ⊢ subst-zero M ↓ γ`, then `γ ⊑ (δ , w) × δ ⊢ M ↓ w` for some `w`.

```
subst-zero-reflect : ∀ {Δ} {δ : Env Δ} {γ : Env (Δ , ⋆)} {M : Δ ⊢ ⋆}
  → δ `⊢ subst-zero M ↓ γ
    ---------------------------------------
  → Σ[ w ∈ Value ] γ `⊑ (δ `, w) × δ ⊢ M ↓ w
subst-zero-reflect {δ = δ} {γ = γ} δσγ = ⟨ last γ , ⟨ lemma , δσγ Z ⟩ ⟩
  where
  lemma : γ `⊑ (δ `, last γ)
  lemma Z = ⊑-refl
  lemma (S x) = var-inv (δσγ (S x))
```

We choose `w` to be the last value in `γ` and we obtain `δ ⊢ M ↓ w` by applying the premise to variable `Z`. Finally, to prove `γ ⊑ (δ , w)`, we prove a lemma by induction in the input variable. The base case is trivial because of our choice of `w`. In the induction case, `S x`, the premise `δ ⊢ subst-zero M ↓ γ` gives us `δ ⊢ x ↓ γ (S x)` and then using `var-inv` we conclude that `γ (S x) ⊑ (δ , w) (S x)`.

Now to prove that substitution reflects denotations.

```
substitution-reflect : ∀ {Δ} {δ : Env Δ} {N : Δ , ⋆ ⊢ ⋆} {M : Δ ⊢ ⋆} {v}
  → δ ⊢ N [ M ] ↓ v
    ------------------------------------------------
  → Σ[ w ∈ Value ] δ ⊢ M ↓ w × (δ `, w) ⊢ N ↓ v
substitution-reflect d with subst-reflect d refl
... | ⟨ γ , ⟨ δσγ , γNv ⟩ ⟩ with subst-zero-reflect δσγ
... | ⟨ w , ⟨ ineq , δMw ⟩ ⟩ = ⟨ w , ⟨ δMw , ⊑-env γNv ineq ⟩ ⟩
```

We apply the `subst-reflect` lemma to obtain `δ ⊢ subst-zero M ↓ γ` and `γ ⊢ N ↓ v` for some `γ`. Using the former, the `subst-zero-reflect` lemma gives us `γ ⊑ (δ , w)` and `δ ⊢ M ↓ w`. We conclude that `δ , w ⊢ N ↓ v` by applying the `⊑-env` lemma, using `γ ⊢ N ↓ v` and `γ ⊑ (δ , w)`.

Reduction reflects denotations

Now that we have proved that substitution reflects denotations, we can easily prove that reduction does too.

```
reflect-beta : ∀{Γ}{γ : Env Γ}{M N}{v}
   → γ ⊢ (N [ M ]) ↓ v
   → γ ⊢ (ƛ N) · M ↓ v
reflect-beta d
   with substitution-reflect d
... | ⟨ v₂′ , ⟨ d₁′ , d₂′ ⟩ ⟩ = ↦-elim (↦-intro d₂′) d₁′

reflect : ∀ {Γ} {γ : Env Γ} {M M′ N v}
  → γ ⊢ N ↓ v → M ⟶ M′ → M′ ≡ N
    ---------------------------------
  → γ ⊢ M ↓ v
reflect var (ξ₁ r) ()
reflect var (ξ₂ r) ()
reflect{γ = γ} (var{x = x}) β mn
```

```
    with var{γ = γ}{x = x}
... | d′ rewrite sym mn = reflect-beta d′
reflect var (ζ r) ()
reflect (↦-elim d₁ d₂) (ξ₁ r) refl = ↦-elim (reflect d₁ r refl) d₂
reflect (↦-elim d₁ d₂) (ξ₂ r) refl = ↦-elim d₁ (reflect d₂ r refl)
reflect (↦-elim d₁ d₂) β mn
    with ↦-elim d₁ d₂
... | d′ rewrite sym mn = reflect-beta d′
reflect (↦-elim d₁ d₂) (ζ r) ()
reflect (↦-intro d) (ξ₁ r) ()
reflect (↦-intro d) (ξ₂ r) ()
reflect (↦-intro d) β mn
    with ↦-intro d
... | d′ rewrite sym mn = reflect-beta d′
reflect (↦-intro d) (ζ r) refl = ↦-intro (reflect d r refl)
reflect ⊥-intro r mn = ⊥-intro
reflect (⊔-intro d₁ d₂) r mn rewrite sym mn =
  ⊔-intro (reflect d₁ r refl) (reflect d₂ r refl)
reflect (sub d lt) r mn = sub (reflect d r mn) lt
```

Reduction implies denotational equality

We have proved that reduction both preserves and reflects denotations. Thus, reduction implies denotational equality.

```
reduce-equal : ∀ {Γ} {M : Γ ⊢ ⋆} {N : Γ ⊢ ⋆}
  → M ⟶ N
    ---------
  → ℰ M ≃ ℰ N
reduce-equal {Γ}{M}{N} r γ v =
    ⟨ (λ m → preserve m r) , (λ n → reflect n r refl) ⟩
```

We conclude with the *soundness property*, that multi-step reduction to a lambda abstraction implies denotational equivalence with a lambda abstraction.

```
soundness : ∀{Γ} {M : Γ ⊢ ⋆} {N : Γ , ⋆ ⊢ ⋆}
  → M ⟶↠ ƛ N
    ------------------
  → ℰ M ≃ ℰ (ƛ N)
soundness (.(ƛ _) ∎) γ v = ⟨ (λ x → x) , (λ x → x) ⟩
soundness {Γ} (L ⟶⟨ r ⟩ M⟶↠N) γ v =
  let ih = soundness M⟶↠N in
  let e = reduce-equal r in
  ≃-trans {Γ} e ih γ v
```

Unicode

This chapter uses the following unicode:

```
≟  U+225F  QUESTIONED EQUAL TO (\?=)
```

Chapter 23

Adequacy: Adequacy of denotational semantics with respect to operational semantics

```
module plfa.part3.Adequacy where
```

Introduction

Having proved a preservation property in the last chapter, a natural next step would be to prove progress. That is, to prove a property of the form

```
If ∅ ⊢ M ↓ v, then either M is a lambda abstraction or M ⟶ N for some N.
```

Such a property would tell us that having a denotation implies either reduction to normal form or divergence. This is indeed true, but we can prove a much stronger property! In fact, having a denotation that is a function value (not `⊥`) implies reduction to a lambda abstraction.

This stronger property, reformulated a bit, is known as *adequacy*. That is, if a term `M` is denotationally equal to a lambda abstraction, then `M` reduces to a lambda abstraction.

```
ℰ M ≃ ℰ (ƛ N)  implies M —↠ ƛ N' for some N'
```

Recall that `ℰ M ≃ ℰ (ƛ N)` is equivalent to saying that `γ ⊢ M ↓ (v ↦ w)` for some `v` and `w`. We will show that `γ ⊢ M ↓ (v ↦ w)` implies multi-step reduction a lambda abstraction. The recursive structure of the derivations for `γ ⊢ M ↓ (v ↦ w)` are completely different from the structure of multi-step reductions, so a direct proof would be challenging. However, The structure of `γ ⊢ M ↓ (v ↦ w)` closer to that of BigStep call-by-name evaluation. Further, we already proved that big-step evaluation implies multi-step reduction to a lambda (`cbn→reduce`). So we shall prove that `γ ⊢ M ↓ (v ↦ w)` implies that `γ' ⊢ M ⇓ c`, where `c` is a closure (a term paired with an environment), `γ'` is an environment that maps variables to closures, and `γ` and `γ'` are appropriate related. The proof will be an induction on the derivation of `γ ⊢ M ↓ v`, and to strengthen the induction hypothesis, we will relate semantic values to closures using a *logical relation* `𝕍`.

The rest of this chapter is organized as follows.

- To make the `𝕍` relation down-closed with respect to `⊑`, we must loosen the requirement that `M` result in a function value and instead require that `M` result in a value that is greater than or equal to a function value. We establish several properties about being "greater than a function''.
- We define the logical relation `𝕍` that relates values and closures, and extend it to a relation on terms `𝔼` and environments `𝔾`. We prove several lemmas that culminate in the property that if `𝕍 v c` and `v′ ⊑ v`, then `𝕍 v′ c`.
- We prove the main lemma, that if `𝔾 γ γ'` and `γ ⊢ M ↓ v`, then `𝔼 v (clos M γ')`.
- We prove adequacy as a corollary to the main lemma.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; _≢_; refl; trans; sym; cong; cong₂; cong-app)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Data.Sum
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Negation using (contradiction)
open import Data.Empty using (⊥-elim) renaming (⊥ to Bot)
open import Data.Unit
open import Relation.Nullary using (Dec; yes; no)
open import Function using (_∘_)
open import plfa.part2.Untyped
    using (Context; _⊢_; ★; _∋_; ∅; _,_; Z; S_; `_; ƛ_; _·_;
           rename; subst; ext; exts; _[_]; subst-zero;
           _—↠_; _—→⟨_⟩_; _∎; _—→_; ξ₁; ξ₂; β; ζ)
open import plfa.part2.Substitution using (ids; sub-id)
open import plfa.part2.BigStep
    using (Clos; clos; ClosEnv; ∅'; _,'_; _⊢_⇓_; ⇓-var; ⇓-lam; ⇓-app; ⇓-determ;
           cbn→reduce)
open import plfa.part3.Denotational
    using (Value; Env; `∅; _`,_; _↦_; _⊑_; _⊢_↓_; ⊥; all-funs∈; _⊔_; ∈→⊑;
           var; ↦-elim; ↦-intro; ⊔-intro; ⊥-intro; sub; 𝒞; _≃_; _iff_;
           ⊑-trans; ⊑-conj-R1; ⊑-conj-R2; ⊑-conj-L; ⊑-refl; ⊑-fun; ⊑-bot; ⊑-dist;
           sub-inv-fun)
open import plfa.part3.Soundness using (soundness)
```

The property of being greater or equal to a function

We define the following short-hand for saying that a value is greater-than or equal to a function value.

```
above-fun : Value → Set
above-fun u = Σ[ v ∈ Value ] Σ[ w ∈ Value ] v ↦ w ⊑ u
```

If a value `u` is greater than a function, then an even greater value `u'` is too.

```
above-fun-⊑ : ∀{u u' : Value}
  → above-fun u → u ⊑ u'
    -------------------
  → above-fun u'
above-fun-⊑ ⟨ v , ⟨ w , lt' ⟩ ⟩ lt = ⟨ v , ⟨ w , ⊑-trans lt' lt ⟩ ⟩
```

The bottom value `⊥` is not greater than a function.

```
above-fun⊥ : ¬ above-fun ⊥
above-fun⊥ ⟨ v , ⟨ w , lt ⟩ ⟩
    with sub-inv-fun lt
... | ⟨ Γ , ⟨ f , ⟨ Γ⊆⊥ , ⟨ lt1 , lt2 ⟩ ⟩ ⟩ ⟩
    with all-funs∈ f
... | ⟨ A , ⟨ B , m ⟩ ⟩
    with Γ⊆⊥ m
... | ()
```

If the join of two values `u` and `u'` is greater than a function, then at least one of them is too.

```
above-fun-⊔ : ∀{u u'}
  → above-fun (u ⊔ u')
  → above-fun u ⊎ above-fun u'
above-fun-⊔{u}{u'} ⟨ v , ⟨ w , v↦w⊑u⊔u' ⟩ ⟩
    with sub-inv-fun v↦w⊑u⊔u'
... | ⟨ Γ , ⟨ f , ⟨ Γ⊆u⊔u' , ⟨ lt1 , lt2 ⟩ ⟩ ⟩ ⟩
    with all-funs∈ f
... | ⟨ A , ⟨ B , m ⟩ ⟩
    with Γ⊆u⊔u' m
... | inj₁ x = inj₁ ⟨ A , ⟨ B , (∈→⊑ x) ⟩ ⟩
... | inj₂ x = inj₂ ⟨ A , ⟨ B , (∈→⊑ x) ⟩ ⟩
```

On the other hand, if neither of `u` and `u'` is greater than a function, then their join is also not greater than a function.

```
not-above-fun-⊔ : ∀{u u' : Value}
  → ¬ above-fun u → ¬ above-fun u'
  → ¬ above-fun (u ⊔ u')
not-above-fun-⊔ naf1 naf2 af12
  with above-fun-⊔ af12
... | inj₁ af1 = contradiction af1 naf1
... | inj₂ af2 = contradiction af2 naf2
```

The converse is also true. If the join of two values is not above a function, then neither of them is individually.

```
not-above-fun-⊔-inv : ∀{u u' : Value} → ¬ above-fun (u ⊔ u')
  → ¬ above-fun u × ¬ above-fun u'
not-above-fun-⊔-inv af = ⟨ f af , g af ⟩
  where
    f : ∀{u u' : Value} → ¬ above-fun (u ⊔ u') → ¬ above-fun u
    f{u}{u'} af12 ⟨ v , ⟨ w , lt ⟩ ⟩ =
      contradiction ⟨ v , ⟨ w , ⊑-conj-R1 lt ⟩ ⟩ af12
    g : ∀{u u' : Value} → ¬ above-fun (u ⊔ u') → ¬ above-fun u'
    g{u}{u'} af12 ⟨ v , ⟨ w , lt ⟩ ⟩ =
      contradiction ⟨ v , ⟨ w , ⊑-conj-R2 lt ⟩ ⟩ af12
```

The property of being greater than a function value is decidable, as exhibited by the following

function.

```
above-fun? : (v : Value) → Dec (above-fun v)
above-fun? ⊥ = no above-fun⊥
above-fun? (v ↦ w) = yes ⟨ v , ⟨ w , ⊑-refl ⟩ ⟩
above-fun? (u ⊔ u')
  with above-fun? u | above-fun? u'
... | yes ⟨ v , ⟨ w , lt ⟩ ⟩ | _ = yes ⟨ v , ⟨ w , (⊑-conj-R1 lt) ⟩ ⟩
... | no _ | yes ⟨ v , ⟨ w , lt ⟩ ⟩ = yes ⟨ v , ⟨ w , (⊑-conj-R2 lt) ⟩ ⟩
... | no x | no y = no (not-above-fun-⊔ x y)
```

Relating values to closures

Next we relate semantic values to closures. The relation `𝕍` is for closures whose term is a lambda abstraction, i.e., in weak-head normal form (WHNF). The relation `𝔼` is for any closure. Roughly speaking, `𝔼 v c` will hold if, when `v` is greater than a function value, `c` evaluates to a closure `c'` in WHNF and `𝕍 v c'`. Regarding `𝕍 v c`, it will hold when `c` is in WHNF, and if `v` is a function, the body of `c` evaluates according to `v`.

```
𝕍 : Value → Clos → Set
𝔼 : Value → Clos → Set
```

We define `𝕍` as a function from values and closures to `Set` and not as a data type because it is mutually recursive with `𝔼` in a negative position (to the left of an implication). We first perform case analysis on the term in the closure. If the term is a variable or application, then `𝕍` is false (`Bot`). If the term is a lambda abstraction, we define `𝕍` by recursion on the value, which we describe below.

```
𝕍 v (clos (` x₁) γ) = Bot
𝕍 v (clos (M · M₁) γ) = Bot
𝕍 ⊥ (clos (ƛ M) γ) = ⊤
𝕍 (v ↦ w) (clos (ƛ N) γ) =
    (∀{c : Clos} → 𝔼 v c → above-fun w → Σ[ c' ∈ Clos ]
      (γ ,' c) ⊢ N ⇓ c' × 𝕍 w c')
𝕍 (u ⊔ v) (clos (ƛ N) γ) = 𝕍 u (clos (ƛ N) γ) × 𝕍 v (clos (ƛ N) γ)
```

- If the value is `⊥`, then the result is true (`⊤`).
- If the value is a join (u ⊔ v), then the result is the pair (conjunction) of `𝕍` is true for both u and v.
- The important case is for a function value `v ↦ w` and closure `clos (ƛ N) γ`. Given any closure `c` such that `𝔼 v c`, if `w` is greater than a function, then `N` evaluates (with `γ` extended with `c`) to some closure `c'` and we have `𝕍 w c'`.

The definition of `𝔼` is straightforward. If `v` is a greater than a function, then `M` evaluates to a closure related to `v`.

```
𝔼 v (clos M γ') = above-fun v → Σ[ c ∈ Clos ] γ' ⊢ M ⇓ c × 𝕍 v c
```

The proof of the main lemma is by induction on `γ ⊢ M ↓ v`, so it goes underneath lambda abstractions and must therefore reason about open terms (terms with variables). So we must relate

environments of semantic values to environments of closures. In the following, `𝔾` relates `γ` to `γ'` if the corresponding values and closures are related by `𝔼`.

```
𝔾 : ∀{Γ} → Env Γ → ClosEnv Γ → Set
𝔾 {Γ} γ γ' = ∀{x : Γ ∋ ⋆} → 𝔼 (γ x) (γ' x)

𝔾-∅ : 𝔾 `∅ ∅'
𝔾-∅ {()}

𝔾-ext : ∀{Γ}{γ : Env Γ}{γ' : ClosEnv Γ}{v c}
      → 𝔾 γ γ' → 𝔼 v c → 𝔾 (γ `, v) (γ' ,' c)
𝔾-ext {Γ} {γ} {γ'} g e {Z} = e
𝔾-ext {Γ} {γ} {γ'} g e {S x} = g
```

We need a few properties of the `𝕍` and `𝔼` relations. The first is that a closure in the `𝕍` relation must be in weak-head normal form. We define WHNF has follows.

```
data WHNF : ∀ {Γ A} → Γ ⊢ A → Set where
  ƛ_ : ∀ {Γ} {N : Γ , ⋆ ⊢ ⋆}
     → WHNF (ƛ N)
```

The proof goes by cases on the term in the closure.

```
𝕍→WHNF : ∀{Γ}{γ : ClosEnv Γ}{M : Γ ⊢ ⋆}{v}
       → 𝕍 v (clos M γ) → WHNF M
𝕍→WHNF {M = ` x} {v} ()
𝕍→WHNF {M = ƛ N} {v} vc = ƛ_
𝕍→WHNF {M = L · M} {v} ()
```

Next we have an introduction rule for `𝕍` that mimics the `⊔-intro` rule. If both `u` and `v` are related to a closure `c`, then their join is too.

```
𝕍⊔-intro : ∀{c u v}
         → 𝕍 u c → 𝕍 v c
           ---------------
         → 𝕍 (u ⊔ v) c
𝕍⊔-intro {clos (` x) γ} () vc
𝕍⊔-intro {clos (ƛ N) γ} uc vc = ⟨ uc , vc ⟩
𝕍⊔-intro {clos (L · M) γ} () vc
```

In a moment we prove that `𝕍` is preserved when going from a greater value to a lesser value: if `𝕍 v c` and `v' ⊑ v`, then `𝕍 v' c`. This property, named `𝕍-sub`, is needed by the main lemma in the case for the `sub` rule.

To prove `𝕍-sub`, we in turn need the following property concerning values that are not greater than a function, that is, values that are equivalent to `⊥`. In such cases, `𝕍 v (clos (ƛ N) γ')` is trivially true.

```
not-above-fun-𝕍 : ∀{v : Value}{Γ}{γ' : ClosEnv Γ}{N : Γ , ⋆ ⊢ ⋆ }
  → ¬ above-fun v
    ------------------
  → 𝕍 v (clos (ƛ N) γ')
not-above-fun-𝕍 {⊥} af = tt
not-above-fun-𝕍 {v ↦ v'} af = ⊥-elim (contradiction ⟨ v , ⟨ v' , ⊑-refl ⟩ ⟩ af)
not-above-fun-𝕍 {v₁ ⊔ v₂} af
  with not-above-fun-⊔-inv af
```

```
... | ⟨ af1 , af2 ⟩ = ⟨ not-above-fun-𝕍 af1 , not-above-fun-𝕍 af2 ⟩
```

The proofs of `𝕍-sub` and `𝔼-sub` are intertwined.

```
sub-𝕍 : ∀{c : Clos}{v v'} → 𝕍 v c → v' ⊑ v → 𝕍 v' c
sub-𝔼 : ∀{c : Clos}{v v'} → 𝔼 v c → v' ⊑ v → 𝔼 v' c
```

We prove `𝕍-sub` by case analysis on the closure's term, to dispatch the cases for variables and application. We then proceed by induction on `v' ⊑ v`. We describe each case below.

```
sub-𝕍 {clos (` x) γ} {v} () lt
sub-𝕍 {clos (L · M) γ} () lt
sub-𝕍 {clos (ƛ N) γ} vc ⊑-bot = tt
sub-𝕍 {clos (ƛ N) γ} vc (⊑-conj-L lt1 lt2) = ⟨ (sub-𝕍 vc lt1) , sub-𝕍 vc lt2 ⟩
sub-𝕍 {clos (ƛ N) γ} ⟨ vv1 , vv2 ⟩ (⊑-conj-R1 lt) = sub-𝕍 vv1 lt
sub-𝕍 {clos (ƛ N) γ} ⟨ vv1 , vv2 ⟩ (⊑-conj-R2 lt) = sub-𝕍 vv2 lt
sub-𝕍 {clos (ƛ N) γ} vc (⊑-trans{v = v₂} lt1 lt2) = sub-𝕍 (sub-𝕍 vc lt2) lt1
sub-𝕍 {clos (ƛ N) γ} vc (⊑-fun lt1 lt2) ev1 sf
    with vc (sub-𝔼 ev1 lt1) (above-fun-⊑ sf lt2)
... | ⟨ c , ⟨ Nc , v4 ⟩ ⟩ = ⟨ c , ⟨ Nc , sub-𝕍 v4 lt2 ⟩ ⟩
sub-𝕍 {clos (ƛ N) γ} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c sf
    with above-fun? w | above-fun? w'
... | yes af2 | yes af3
    with vcw ev1c af2 | vcw' ev1c af3
... | ⟨ clos L δ , ⟨ L⇓c₂ , 𝕍w ⟩ ⟩
    | ⟨ c₃ , ⟨ L⇓c₃ , 𝕍w' ⟩ ⟩ rewrite ⇓-determ L⇓c₃ L⇓c₂ with 𝕍→WHNF 𝕍w
... | ƛ_ =
      ⟨ clos L δ , ⟨ L⇓c₂ , ⟨ 𝕍w , 𝕍w' ⟩ ⟩ ⟩
sub-𝕍 {c} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c sf
    | yes af2 | no naf3
    with vcw ev1c af2
... | ⟨ clos {Γ'} L γ₁ , ⟨ L⇓c2 , 𝕍w ⟩ ⟩
    with 𝕍→WHNF 𝕍w
... | ƛ_ {N = N'} =
      let 𝕍w' = not-above-fun-𝕍{w'}{Γ'}{γ₁}{N'} naf3 in
      ⟨ clos (ƛ N') γ₁ , ⟨ L⇓c2 , 𝕍⊔-intro 𝕍w 𝕍w' ⟩ ⟩
sub-𝕍 {c} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c sf
    | no naf2 | yes af3
    with vcw' ev1c af3
... | ⟨ clos {Γ'} L γ₁ , ⟨ L⇓c3 , 𝕍w'c ⟩ ⟩
    with 𝕍→WHNF 𝕍w'c
... | ƛ_ {N = N'} =
      let 𝕍wc = not-above-fun-𝕍{w}{Γ'}{γ₁}{N'} naf2 in
      ⟨ clos (ƛ N') γ₁ , ⟨ L⇓c3 , 𝕍⊔-intro 𝕍wc 𝕍w'c ⟩ ⟩
sub-𝕍 {c} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c ⟨ v' , ⟨ w'' , lt ⟩ ⟩
    | no naf2 | no naf3
    with above-fun-⊔ ⟨ v' , ⟨ w'' , lt ⟩ ⟩
... | inj₁ af2 = ⊥-elim (contradiction af2 naf2)
... | inj₂ af3 = ⊥-elim (contradiction af3 naf3)
```

- Case `⊑-bot`. We immediately have `𝕍 ⊥ (clos (ƛ N) γ)`.
- Case `⊑-conj-L`.

  ```
      v₁' ⊑ v     v₂' ⊑ v
      -------------------
  ```

```
    (v₁' ⊔ v₂') ⊑ v
```

The induction hypotheses gives us `𝕍 v₁' (clos (ƛ N) γ)` and `𝕍 v₂' (clos (ƛ N) γ)`, which is all we need for this case.

- Case `⊑-conj-R1`.

```
    v' ⊑ v₁
    -------------
    v' ⊑ (v₁ ⊔ v₂)
```

The induction hypothesis gives us `𝕍 v' (clos (ƛ N) γ)`.

- Case `⊑-conj-R2`.

```
    v' ⊑ v₂
    -------------
    v' ⊑ (v₁ ⊔ v₂)
```

Again, the induction hypothesis gives us `𝕍 v' (clos (ƛ N) γ)`.

- Case `⊑-trans`.

```
    v' ⊑ v₂   v₂ ⊑ v
    -----------------
         v' ⊑ v
```

The induction hypothesis for `v₂ ⊑ v` gives us `𝕍 v₂ (clos (ƛ N) γ)`. We apply the induction hypothesis for `v' ⊑ v₂` to conclude that `𝕍 v' (clos (ƛ N) γ)`.

- Case `⊑-dist`. This case is the most difficult. We have

```
    𝕍 (v ↦ w) (clos (ƛ N) γ)
    𝕍 (v ↦ w') (clos (ƛ N) γ)
```

and need to show that

```
    𝕍 (v ↦ (w ⊔ w')) (clos (ƛ N) γ)
```

Let `c` be an arbitrary closure such that `𝔼 v c`. Assume `w ⊔ w'` is greater than a function. Unfortunately, this does not mean that both `w` and `w'` are above functions. But thanks to the lemma `above-fun-⊔`, we know that at least one of them is greater than a function.

  - Suppose both of them are greater than a function. Then we have `γ ⊢ N ⇓ clos L δ` and `𝕍 w (clos L δ)`. We also have `γ ⊢ N ⇓ c₃` and `𝕍 w' c₃`. Because the big-step semantics is deterministic, we have `c₃ ≡ clos L δ`. Also, from `𝕍 w (clos L δ)` we know that `L ≡ ƛ N'` for some `N'`. We conclude that `𝕍 (w ⊔ w') (clos (ƛ N') δ)`.
  - Suppose one of them is greater than a function and the other is not: say `above-fun w` and `¬ above-fun w'`. Then from `𝕍 (v ↦ w) (clos (ƛ N) γ)` we have `γ ⊢ N ⇓ clos L γ₁` and `𝕍 w (clos L γ₁)`. From this we have `L ≡ ƛ N'` for some `N'`. Meanwhile, from `¬ above-fun w'` we have `𝕍 w' (clos L γ₁)`. We conclude that `𝕍 (w ⊔ w') (clos (ƛ N') γ₁)`.

The proof of `sub-𝔼` is direct and explained below.

```
sub-𝔼 {clos M γ} {v} {v'} 𝔼v v'⊑v fv'
  with 𝔼v (above-fun-⊑ fv' v'⊑v)
... | ⟨ c , ⟨ M⇓c , 𝕍v ⟩ ⟩ =
      ⟨ c , ⟨ M⇓c , sub-𝕍 𝕍v v'⊑v ⟩ ⟩
```

From `above-fun v'` and `v' ⊑ v` we have `above-fun v`. Then with `𝔼 v c` we obtain a closure `c` such that `γ ⊢ M ⇓ c` and `𝕍 v c`. We conclude with an application of `sub-𝕍` with `v' ⊑ v` to show `𝕍 v' c`.

Programs with function denotation terminate via call-by-name

The main lemma proves that if a term has a denotation that is above a function, then it terminates via call-by-name. More formally, if `γ ⊢ M ↓ v` and `𝔾 γ γ'`, then `𝔼 v (clos M γ')`. The proof is by induction on the derivation of `γ ⊢ M ↓ v` we discuss each case below.

The following lemma, kth-x, is used in the case for the `var` rule.

```
kth-x : ∀{Γ}{γ' : ClosEnv Γ}{x : Γ ∋ ⋆}
  → Σ[ Δ ∈ Context ] Σ[ δ ∈ ClosEnv Δ ] Σ[ M ∈ Δ ⊢ ⋆ ]
            γ' x ≡ clos M δ
kth-x{γ' = γ'}{x = x} with γ' x
... | clos{Γ = Δ} M δ = ⟨ Δ , ⟨ δ , ⟨ M , refl ⟩ ⟩ ⟩
```

```
↓→𝔼 : ∀{Γ}{γ : Env Γ}{γ' : ClosEnv Γ}{M : Γ ⊢ ⋆ }{v}
           → 𝔾 γ γ' → γ ⊢ M ↓ v → 𝔼 v (clos M γ')
↓→𝔼 {Γ} {γ} {γ'} 𝔾γγ' (var{x = x}) fγx
    with kth-x{Γ}{γ'}{x} | 𝔾γγ'{x = x}
... | ⟨ Δ , ⟨ δ , ⟨ M' , eq ⟩ ⟩ ⟩ | 𝔾γγ'x rewrite eq
    with 𝔾γγ'x fγx
... | ⟨ c , ⟨ M'⇓c , 𝕍γx ⟩ ⟩ =
      ⟨ c , ⟨ (⇓-var eq M'⇓c) , 𝕍γx ⟩ ⟩
↓→𝔼 {Γ} {γ} {γ'} 𝔾γγ' (↦-elim{L = L}{M = M}{v = v₁}{w = v} d₁ d₂) fv
    with ↓→𝔼 𝔾γγ' d₁ ⟨ v₁ , ⟨ v , ⊑-refl ⟩ ⟩
... | ⟨ clos L' δ , ⟨ L⇓L' , 𝕍v₁↦v ⟩ ⟩
    with 𝕍→WHNF 𝕍v₁↦v
... | ƛ_ {N = N}
    with 𝕍v₁↦v {clos M γ'} (↓→𝔼 𝔾γγ' d₂) fv
... | ⟨ c' , ⟨ N⇓c' , 𝕍v ⟩ ⟩ =
    ⟨ c' , ⟨ ⇓-app L⇓L' N⇓c' , 𝕍v ⟩ ⟩
↓→𝔼 {Γ} {γ} {γ'} 𝔾γγ' (↦-intro{N = N}{v = v}{w = w} d) fv↦w =
    ⟨ clos (ƛ N) γ' , ⟨ ⇓-lam , E ⟩ ⟩
    where E : {c : Clos} → 𝔼 v c → above-fun w
              → Σ[ c' ∈ Clos ] (γ' ,' c) ⊢ N ⇓ c' × 𝕍 w c'
          E {c} 𝔼vc fw = ↓→𝔼 (λ {x} → 𝔾-ext{Γ}{γ}{γ'} 𝔾γγ' 𝔼vc {x}) d fw
↓→𝔼 𝔾γγ' ⊥-intro f⊥ = ⊥-elim (above-fun⊥ f⊥)
↓→𝔼 𝔾γγ' (⊔-intro{v = v₁}{w = v₂} d₁ d₂) fv12
    with above-fun? v₁ | above-fun? v₂
... | yes fv1 | yes fv2
    with ↓→𝔼 𝔾γγ' d₁ fv1 | ↓→𝔼 𝔾γγ' d₂ fv2
... | ⟨ c₁ , ⟨ M⇓c₁ , 𝕍v₁ ⟩ ⟩ | ⟨ c₂ , ⟨ M⇓c₂ , 𝕍v₂ ⟩ ⟩
    rewrite ⇓-determ M⇓c₂ M⇓c₁ =
    ⟨ c₁ , ⟨ M⇓c₁ , 𝕍⊔-intro 𝕍v₁ 𝕍v₂ ⟩ ⟩
```

```
↓→𝔼 𝔾γγ' (⊔-intro{v = v₁}{w = v₂} d₁ d₂) fv12 | yes fv1 | no nfv2
    with ↓→𝔼 𝔾γγ' d₁ fv1
... | ⟨ clos {Γ'} M' γ₁ , ⟨ M⇓c₁ , 𝕍v₁ ⟩ ⟩
    with 𝕍→WHNF 𝕍v₁
... | ƛ_ {N = N} =
    let 𝕍v₂ = not-above-fun-𝕍{v₂}{Γ'}{γ₁}{N} nfv2 in
    ⟨ clos (ƛ N) γ₁ , ⟨ M⇓c₁ , 𝕍⊔-intro 𝕍v₁ 𝕍v₂ ⟩ ⟩
↓→𝔼 𝔾γγ' (⊔-intro{v = v₁}{w = v₂} d₁ d₂) fv12 | no nfv1 | yes fv2
    with ↓→𝔼 𝔾γγ' d₂ fv2
... | ⟨ clos {Γ'} M' γ₁ , ⟨ M'⇓c₂ , 𝕍2c ⟩ ⟩
    with 𝕍→WHNF 𝕍2c
... | ƛ_ {N = N} =
    let 𝕍1c = not-above-fun-𝕍{v₁}{Γ'}{γ₁}{N} nfv1 in
    ⟨ clos (ƛ N) γ₁ , ⟨ M'⇓c₂ , 𝕍⊔-intro 𝕍1c 𝕍2c ⟩ ⟩
↓→𝔼 𝔾γγ' (⊔-intro d₁ d₂) fv12 | no nfv1 | no nfv2
    with above-fun-⊔ fv12
... | inj₁ fv1 = ⊥-elim (contradiction fv1 nfv1)
... | inj₂ fv2 = ⊥-elim (contradiction fv2 nfv2)
↓→𝔼 {Γ} {γ} {γ'} {M} {v'} 𝔾γγ' (sub{v = v} d v'⊑v) fv'
    with ↓→𝔼 {Γ} {γ} {γ'} {M} 𝔾γγ' d (above-fun-⊑ fv' v'⊑v)
... | ⟨ c , ⟨ M⇓c , 𝕍v ⟩ ⟩ =
      ⟨ c , ⟨ M⇓c , sub-𝕍 𝕍v v'⊑v ⟩ ⟩
```

- Case `var`. Looking up `x` in `γ'` yields some closure, `clos M' δ`, and from `𝔾 γ γ'` we have `𝔼 (γ x) (clos M' δ)`. With the premise `above-fun (γ x)`, we obtain a closure `c` such that `δ ⊢ M' ⇓ c` and `𝕍 (γ x) c`. To conclude `γ' ⊢ x ⇓ c` via `⇓-var`, we need `γ' x ≡ clos M' δ`, which is obvious, but it requires some Agda shananigans via the `kth-x` lemma to get our hands on it.
- Case `↦-elim`. We have `γ ⊢ L · M ↓ v`. The induction hypothesis for `γ ⊢ L ↓ v₁ ↦ v` gives us `γ' ⊢ L ⇓ clos L' δ` and `𝕍 v (clos L' δ)`. Of course, `L' ≡ ƛ N` for some `N`. By the induction hypothesis for `γ ⊢ M ↓ v₁`, we have `𝔼 v₁ (clos M γ')`. Together with the premise `above-fun v` and `𝕍 v (clos L' δ)`, we obtain a closure `c'` such that `δ ⊢ N ⇓ c'` and `𝕍 v c'`. We conclude that `γ' ⊢ L · M ⇓ c'` by rule `⇓-app`.
- Case `↦-intro`. We have `γ ⊢ ƛ N ↓ v ↦ w`. We immediately have `γ' ⊢ ƛ M ⇓ clos (ƛ M) γ'` by rule `⇓-lam`. But we also need to prove `𝕍 (v ↦ w) (clos (ƛ N) γ')`. Let `c` by an arbitrary closure such that `𝔼 v c`. Suppose `v'` is greater than a function value. We need to show that `γ' , c ⊢ N ⇓ c'` and `𝕍 v' c'` for some `c'`. We prove this by the induction hypothesis for `γ , v ⊢ N ↓ v'` but we must first show that `𝔾 (γ , v) (γ' , c)`. We prove that by the lemma `𝔾-ext`, using facts `𝔾 γ γ'` and `𝔼 v c`.
- Case `⊥-intro`. We have the premise `above-fun ⊥`, but that's impossible.
- Case `⊔-intro`. We have `γ ⊢ M ↓ (v₁ ⊔ v₂)` and `above-fun (v₁ ⊔ v₂)` and need to show `γ' ⊢ M ↓ c` and `𝕍 (v₁ ⊔ v₂) c` for some `c`. Again, by `above-fun-⊔`, at least one of `v₁` or `v₂` is greater than a function.
  - Suppose both `v₁` and `v₂` are greater than a function value. By the induction hypotheses for `γ ⊢ M ↓ v₁` and `γ ⊢ M ↓ v₂` we have `γ' ⊢ M ⇓ c₁`, `𝕍 v₁ c₁`, `γ' ⊢ M ⇓ c₂`, and `𝕍 v₂ c₂` for some `c₁` and `c₂`. Because `⇓` is deterministic, we have `c₂ ≡ c₁`. Then by `𝕍⊔-intro` we conclude that `𝕍 (v₁ ⊔ v₂) c₁`.

  - Without loss of generality, suppose `v₁` is greater than a function value but `v₂` is not. By the induction hypotheses for `γ ⊢ M ↓ v₁`, and using `𝕍→WHNF`, we have `γ' ⊢ M ⇓ clos (ƛ N) γ₁` and `𝕍 v₁ (clos (ƛ N) γ₁)`. Then because `v₂` is not greater than a function, we also have `𝕍 v₂ (clos (ƛ N) γ₁)`. We conclude that `𝕍 (v₁ ⊔ v₂) (clos (ƛ N) γ₁)`.

- Case `sub`. We have `γ ⊢ M ↓ v`, `v' ⊑ v`, and `above-fun v'`. We need to show that `γ' ⊢ M ⇓ c` and `𝕍 v' c` for some `c`. We have `above-fun v` by `above-fun-⊑`, so the induction hypothesis for `γ ⊢ M ↓ v` gives us a closure `c` such that `γ' ⊢ M ⇓ c` and `𝕍 v c`. We conclude that `𝕍 v' c` by `sub-𝕍`.

Proof of denotational adequacy

From the main lemma we can directly show that `ℰ M ≃ ℰ (ƛ N)` implies that `M` big-steps to a lambda, i.e., `∅ ⊢ M ⇓ clos (ƛ N′) γ`.

```
↓→⇓ : ∀{M : ∅ ⊢ ★}{N : ∅ , ★ ⊢ ★} → ℰ M ≃ ℰ (ƛ N)
       → Σ[ Γ ∈ Context ] Σ[ N′ ∈ (Γ , ★ ⊢ ★) ] Σ[ γ ∈ ClosEnv Γ ]
         ∅' ⊢ M ⇓ clos (ƛ N′) γ
↓→⇓{M}{N} eq
    with ↓→𝔼 𝔾-∅ ((proj₂ (eq `∅ (⊥ ↦ ⊥))) (↦-intro ⊥-intro))
                 ⟨ ⊥ , ⟨ ⊥ , ⊑-refl ⟩ ⟩
... | ⟨ clos {Γ} M′ γ , ⟨ M⇓c , Vc ⟩ ⟩
    with 𝕍→WHNF Vc
... | ƛ_ {N = N′} =
    ⟨ Γ , ⟨ N′ , ⟨ γ , M⇓c ⟩ ⟩ ⟩
```

The proof goes as follows. We derive `∅ ⊢ ƛ N ↓ ⊥ ↦ ⊥` and then `ℰ M ≃ ℰ (ƛ N)` gives us `∅ ⊢ M ↓ ⊥ ↦ ⊥`. We conclude by applying the main lemma to obtain `∅ ⊢ M ⇓ clos (ƛ N′) γ` for some `N′` and `γ`.

Now to prove the adequacy property. We apply the above lemma to obtain `∅ ⊢ M ⇓ clos (ƛ N′) γ` and then apply `cbn→reduce` to conclude.

```
adequacy : ∀{M : ∅ ⊢ ★}{N : ∅ , ★ ⊢ ★}
  → ℰ M ≃ ℰ (ƛ N)
  → Σ[ N′ ∈ (∅ , ★ ⊢ ★) ]
    (M —↠ ƛ N′)
adequacy{M}{N} eq
    with ↓→⇓ eq
... | ⟨ Γ , ⟨ N′ , ⟨ γ , M⇓ ⟩ ⟩ ⟩ =
    cbn→reduce M⇓
```

Call-by-name is equivalent to beta reduction

As promised, we return to the question of whether call-by-name evaluation is equivalent to beta reduction. In chapter BigStep we established the forward direction: that if call-by-name produces a result, then the program beta reduces to a lambda abstraction (`cbn→reduce`). We now prove the backward direction of the if-and-only-if, leveraging our results about the denotational semantics.

```
reduce→cbn : ∀ {M : ∅ ⊢ ★} {N : ∅ , ★ ⊢ ★}
           → M —↠ ƛ N
           → Σ[ Δ ∈ Context ] Σ[ N′ ∈ Δ , ★ ⊢ ★ ] Σ[ δ ∈ ClosEnv Δ ]
             ∅' ⊢ M ⇓ clos (ƛ N′) δ
reduce→cbn M—↠ƛN = ↓→⇓ (soundness M—↠ƛN)
```

Suppose `M —↠ ƛ N`. Soundness of the denotational semantics gives us `ℰ M ≃ ℰ (ƛ N)`. Then by `↓→⇓` we conclude that `∅' ⊢ M ⇓ clos (ƛ N′) δ` for some `N′` and `δ`.

Putting the two directions of the if-and-only-if together, we establish that call-by-name evaluation is equivalent to beta reduction in the following sense.

```
cbn↔reduce : ∀ {M : ∅ ⊢ ★}
           → (Σ[ N ∈ ∅ , ★ ⊢ ★ ] (M —↠ ƛ N))
             iff
             (Σ[ Δ ∈ Context ] Σ[ N′ ∈ Δ , ★ ⊢ ★ ] Σ[ δ ∈ ClosEnv Δ ]
               ∅' ⊢ M ⇓ clos (ƛ N′) δ)
cbn↔reduce {M} = ⟨ (λ x → reduce→cbn (proj₂ x)) ,
                   (λ x → cbn→reduce (proj₂ (proj₂ (proj₂ x)))) ⟩
```

Unicode

This chapter uses the following unicode:

```
𝔼  U+1D53C  MATHEMATICAL DOUBLE-STRUCK CAPITAL E (\bE)
𝔾  U+1D53E  MATHEMATICAL DOUBLE-STRUCK CAPITAL G (\bG)
𝕍  U+1D53E  MATHEMATICAL DOUBLE-STRUCK CAPITAL V (\bV)
```

Chapter 24

ContextualEquivalence: Denotational equality implies contextual equivalence

```
module plfa.part3.ContextualEquivalence where
```

Imports

```
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
    renaming (_,_ to ⟨_,_⟩)
open import plfa.part2.Untyped using (_⊢_; ★; ∅; _,_; ƛ_; _—↠_)
open import plfa.part2.BigStep using (_⊢_⇓_; cbn→reduce)
open import plfa.part3.Denotational using (𝒞; _≃_; ≃-sym; ≃-trans; _iff_)
open import plfa.part3.Compositional using (Ctx; plug; compositionality)
open import plfa.part3.Soundness using (soundness)
open import plfa.part3.Adequacy using (↓→⇓)
```

Contextual Equivalence

The notion of *contextual equivalence* is an important one for programming languages because it is the sufficient condition for changing a subterm of a program while maintaining the program's overall behavior. Two terms `M` and `N` are contextually equivalent if they can plugged into any context `C` and produce equivalent results. As discuss in the Denotational chapter, the result of a program in the lambda calculus is to terminate or not. We characterize termination with the reduction semantics as follows.

```
terminates : ∀{Γ} → (M : Γ ⊢ ★) → Set
terminates {Γ} M = Σ[ N ∈ (Γ , ★ ⊢ ★) ] (M —↠ ƛ N)
```

So two terms are contextually equivalent if plugging them into the same context produces two programs that either terminate or diverge together.

```
_≅_ : ∀{Γ} → (M N : Γ ⊢ ★) → Set
(_≅_ {Γ} M N) = ∀ {C : Ctx Γ ∅}
              → (terminates (plug C M)) iff (terminates (plug C N))
```

The contextual equivalence of two terms is difficult to prove directly based on the above definition because of the universal quantification of the context `C`. One of the main motivations for developing denotational semantics is to have an alternative way to prove contextual equivalence that instead only requires reasoning about the two terms.

Denotational equivalence implies contextual equivalence

Thankfully, the proof that denotational equality implies contextual equivalence is an easy corollary of the results that we have already established. Furthermore, the two directions of the if-and-only-if are symmetric, so we can prove one lemma and then use it twice in the theorem.

The lemma states that if `M` and `N` are denotationally equal and if `M` plugged into `C` terminates, then so does `N` plugged into `C`.

```
denot-equal-terminates : ∀{Γ} {M N : Γ ⊢ ★} {C : Ctx Γ ∅}
  → ℰ M ≃ ℰ N → terminates (plug C M)
    ---------------------------------
  → terminates (plug C N)
denot-equal-terminates {Γ}{M}{N}{C} ℰM≃ℰN ⟨ N′ , CM—↠𝒱N′ ⟩ =
  let ℰCM≃ℰ𝒱N′ = soundness CM—↠𝒱N′ in
  let ℰCM≃ℰCN = compositionality{Γ = Γ}{Δ = ∅}{C = C} ℰM≃ℰN in
  let ℰCN≃ℰ𝒱N′ = ≃-trans (≃-sym ℰCM≃ℰCN) ℰCM≃ℰ𝒱N′ in
    cbn→reduce (proj₂ (proj₂ (proj₂ (↓→⇓ ℰCN≃ℰ𝒱N′))))
```

The proof is direct. Because `plug C —↠ plug C (𝒱N′)`, we can apply soundness to obtain

```
ℰ (plug C M) ≃ ℰ (𝒱N′)
```

From `ℰ M ≃ ℰ N`, compositionality gives us

```
ℰ (plug C M) ≃ ℰ (plug C N).
```

Putting these two facts together gives us

```
ℰ (plug C N) ≃ ℰ (𝒱N′).
```

We then apply `↓→⇓` from Chapter Adequacy to deduce

```
∅' ⊢ plug C N ⇓ clos (𝒱 N′′) δ).
```

Call-by-name evaluation implies reduction to a lambda abstraction, so we conclude that

```
terminates (plug C N).
```

The main theorem follows by two applications of the lemma.

```
denot-equal-contex-equal : ∀{Γ} {M N : Γ ⊢ ★}
  → ℰ M ≃ ℰ N
    ---------
```

```
  → M ≅ N
denot-equal-contex-equal{Γ}{M}{N} eq {C} =
  ⟨ (λ tm → denot-equal-terminates eq tm) ,
    (λ tn → denot-equal-terminates (≃-sym eq) tn) ⟩
```

Unicode

This chapter uses the following unicode:

```
≅  U+2245  APPROXIMATELY EQUAL TO (\~= or \cong)
```

Part IV

Appendix

Appendix A

Substitution: Substitution in the untyped lambda calculus

```
module plfa.part2.Substitution where
```

Introduction

The primary purpose of this chapter is to prove that substitution commutes with itself. Barendgredt (1984) refers to this as the substitution lemma:

```
M [x:=N] [y:=L] = M [y:=L] [x:= N[y:=L] ]
```

In our setting, with de Bruijn indices for variables, the statement of the lemma becomes:

```
M [ N ] [ L ] ≡  M⟮ L ⟯[ N [ L ] ]                    (substitution)
```

where the notation `M ⟮ L ⟯` is for substituting L for index 1 inside M. In addition, because we define substitution in terms of parallel substitution, we have the following generalization, replacing the substitution of L with an arbitrary parallel substitution σ.

```
subst σ (M [ N ]) ≡ (subst (exts σ) M) [ subst σ N ]    (subst-commute)
```

The special case for renamings is also useful.

```
rename ρ (M [ N ]) ≡ (rename (ext ρ) M) [ rename ρ N ]
                                                (rename-subst-commute)
```

The secondary purpose of this chapter is to define the σ algebra of parallel substitution due to Abadi, Cardelli, Curien, and Levy (1991). The equations of this algebra not only help us prove the substitution lemma, but they are generally useful. Furthermore, when the equations are applied from left to right, they form a rewrite system that *decides* whether any two substitutions are equal.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; cong; cong₂; cong-app)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _∎)
open import Function using (_∘_)
open import plfa.part2.Untyped
    using (Type; Context; _⊢_; ★; _∋_; ∅; _,_; Z; S_; `_; ↑_; _·_;
           rename; subst; ext; exts; _[_]; subst-zero)
```

```
postulate
  extensionality : ∀ {A B : Set} {f g : A → B}
    → (∀ (x : A) → f x ≡ g x)
      -----------------------
    → f ≡ g
```

Notation

We introduce the following shorthand for the type of a *renaming* from variables in context `Γ` to variables in context `Δ`.

```
Rename : Context → Context → Set
Rename Γ Δ = ∀{A} → Γ ∋ A → Δ ∋ A
```

Similarly, we introduce the following shorthand for the type of a *substitution* from variables in context `Γ` to terms in context `Δ`.

```
Subst : Context → Context → Set
Subst Γ Δ = ∀{A} → Γ ∋ A → Δ ⊢ A
```

We use the following more succinct notation the `subst` function.

```
⟪_⟫ : ∀{Γ Δ A} → Subst Γ Δ → Γ ⊢ A → Δ ⊢ A
⟪ σ ⟫ = λ M → subst σ M
```

The σ algebra of substitution

Substitutions map de Bruijn indices (natural numbers) to terms, so we can view a substitution simply as a sequence of terms, or more precisely, as an infinite sequence of terms. The σ algebra consists of four operations for building such sequences: identity `ids`, shift `↑`, cons `M • σ`, and sequencing `σ ⨟ τ`. The sequence `0, 1, 2, ...` is constructed by the identity substitution.

```
ids : ∀{Γ} → Subst Γ Γ
ids x = ` x
```

The shift operation `↑` constructs the sequence

```
1, 2, 3, ...
```

and is defined as follows.

```
↑ : ∀{Γ A} → Subst Γ (Γ , A)
↑ x = ` (S x)
```

Given a term `M` and substitution `σ`, the operation `M • σ` constructs the sequence

```
M , σ 0, σ 1, σ 2, ...
```

This operation is analogous to the `cons` operation of Lisp.

```
infixr 6 _•_

_•_ : ∀{Γ Δ A} → (Δ ⊢ A) → Subst Γ Δ → Subst (Γ , A) Δ
(M • σ) Z = M
(M • σ) (S x) = σ x
```

Given two substitutions `σ` and `τ`, the sequencing operation `σ ⨟ τ` produces the sequence

```
⟪τ⟫(σ 0), ⟪τ⟫(σ 1), ⟪τ⟫(σ 2), ...
```

That is, it composes the two substitutions by first applying `σ` and then applying `τ`.

```
infixr 5 _⨟_

_⨟_ : ∀{Γ Δ Σ} → Subst Γ Δ → Subst Δ Σ → Subst Γ Σ
σ ⨟ τ = ⟪ τ ⟫ ∘ σ
```

For the sequencing operation, Abadi et al. use the notation of function composition, writing `σ ∘ τ`, but still with `σ` applied before `τ`, which is the opposite of standard mathematical practice. We instead write `σ ⨟ τ`, because semicolon is the standard notation for forward function composition.

The σ algebra equations

The σ algebra includes the following equations.

```
(sub-head)  ⟪ M • σ ⟫ (` Z) ≡ M
(sub-tail)  ↑ ⨟ (M • σ)     ≡ σ
(sub-η)     (⟪ σ ⟫ (` Z)) • (↑ ⨟ σ) ≡ σ
(Z-shift)   (` Z) • ↑       ≡ ids

(sub-id)    ⟪ ids ⟫ M       ≡ M
(sub-app)   ⟪ σ ⟫ (L · M)   ≡ (⟪ σ ⟫ L) · (⟪ σ ⟫ M)
(sub-abs)   ⟪ σ ⟫ (ƛ N)     ≡ ƛ ⟪ σ ⟫ N
(sub-sub)   ⟪ τ ⟫ ⟪ σ ⟫ M   ≡ ⟪ σ ⨟ τ ⟫ M

(sub-idL)   ids ⨟ σ         ≡ σ
(sub-idR)   σ ⨟ ids         ≡ σ
(sub-assoc) (σ ⨟ τ) ⨟ θ     ≡ σ ⨟ (τ ⨟ θ)
(sub-dist)  (M • σ) ⨟ τ     ≡ (⟪ τ ⟫ M) • (σ ⨟ τ)
```

The first group of equations describe how the `•` operator acts like cons. The equation `sub-head` says that the variable zero `Z` returns the head of the sequence (it acts like the `car` of Lisp). Similarly, `sub-tail` says that sequencing with shift `↑` returns the tail of the sequence (it acts

like `cdr` of Lisp). The `sub-η` equation is the η-expansion rule for sequences, saying that taking the head and tail of a sequence, and then cons'ing them together yields the original sequence. The `Z-shift` equation says that cons'ing zero onto the shifted sequence produces the identity sequence.

The next four equations involve applying substitutions to terms. The equation `sub-id` says that the identity substitution returns the term unchanged. The equations `sub-app` and `sub-abs` says that substitution is a congruence for the lambda calculus. The `sub-sub` equation says that the sequence operator `⨟` behaves as intended.

The last four equations concern the sequencing of substitutions. The first two equations, `sub-idL` and `sub-idR`, say that `ids` is the left and right unit of the sequencing operator. The `sub-assoc` equation says that sequencing is associative. Finally, `sub-dist` says that post-sequencing distributes through cons.

Relating the σ algebra and substitution functions

The definitions of substitution `N [ M ]` and parallel substitution `subst σ N` depend on several auxiliary functions: `rename`, `exts`, `ext`, and `subst-zero`. We shall relate those functions to terms in the σ algebra.

To begin with, renaming can be expressed in terms of substitution. We have

```
rename ρ M ≡ ⟪ ren ρ ⟫ M                (rename-subst-ren)
```

where `ren` turns a renaming `ρ` into a substitution by post-composing `ρ` with the identity substitution.

```
ren : ∀{Γ Δ} → Rename Γ Δ → Subst Γ Δ
ren ρ = ids ∘ ρ
```

When the renaming is the increment function, then it is equivalent to shift.

```
ren S_ ≡ ↑                             (ren-shift)

rename S_ M ≡ ⟪ ↑ ⟫ M                  (rename-shift)
```

Renaming with the identity renaming leaves the term unchanged.

```
rename (λ {A} x → x) M ≡ M             (rename-id)
```

Next we relate the `exts` function to the σ algebra. Recall that the `exts` function extends a substitution as follows:

```
exts σ = ` Z, rename S_ (σ 0), rename S_ (σ 1), rename S_ (σ 2), ...
```

So `exts` is equivalent to cons'ing Z onto the sequence formed by applying `σ` and then shifting.

```
exts σ ≡ ` Z • (σ ⨟ ↑)                (exts-cons-shift)
```

The `ext` function does the same job as `exts` but for renamings instead of substitutions. So composing `ext` with `ren` is the same as composing `ren` with `exts`.

```
ren (ext ρ) ≡ exts (ren ρ)            (ren-ext)
```

Thus, we can recast the `exts-cons-shift` equation in terms of renamings.

```
ren (ext ρ) ≡ ` Z • (ren ρ ⨟ ↑)       (ext-cons-Z-shift)
```

It is also useful to specialize the `sub-sub` equation of the σ algebra to the situation where the first substitution is a renaming.

```
⟪ σ ⟫ (rename ρ M) ≡ ⟪ σ ∘ ρ ⟫ M       (rename-subst)
```

The `subst-zero M` substitution is equivalent to cons'ing `M` onto the identity substitution.

```
subst-zero M ≡ M • ids                 (subst-Z-cons-ids)
```

Finally, sequencing `exts σ` with `subst-zero M` is equivalent to cons'ing `M` onto `σ`.

```
exts σ ⨟ subst-zero M ≡ (M • σ)        (subst-zero-exts-cons)
```

Proofs of sub-head, sub-tail, sub-η, Z-shift, sub-idL, sub-dist, and sub-app

We start with the proofs that are immediate from the definitions of the operators.

```
sub-head : ∀ {Γ Δ} {A} {M : Δ ⊢ A}{σ : Subst Γ Δ}
         → ⟪ M • σ ⟫ (` Z) ≡ M
sub-head = refl
```

```
sub-tail : ∀{Γ Δ} {A B} {M : Δ ⊢ A} {σ : Subst Γ Δ}
         → (↑ ⨟ M • σ) {A = B} ≡ σ
sub-tail = extensionality λ x → refl
```

```
sub-η : ∀{Γ Δ} {A B} {σ : Subst (Γ , A) Δ}
      → (⟪ σ ⟫ (` Z) • (↑ ⨟ σ)) {A = B} ≡ σ
sub-η {Γ}{Δ}{A}{B}{σ} = extensionality λ x → lemma
  where
  lemma : ∀ {x} → ((⟪ σ ⟫ (` Z)) • (↑ ⨟ σ)) x ≡ σ x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

```
Z-shift : ∀{Γ}{A B}
        → ((` Z) • ↑) ≡ ids {Γ , A} {B}
Z-shift {Γ}{A}{B} = extensionality lemma
  where
  lemma : (x : Γ , A ∋ B) → ((` Z) • ↑) x ≡ ids x
  lemma Z = refl
  lemma (S y) = refl
```

```
sub-idL : ∀{Γ Δ} {σ : Subst Γ Δ} {A}
  → ids ⨟ σ ≡ σ {A}
sub-idL = extensionality λ x → refl
```

```
sub-dist : ∀{Γ Δ Σ : Context} {A B} {σ : Subst Γ Δ} {τ : Subst Δ Σ}
             {M : Δ ⊢ A}
         → ((M • σ) ⨟ τ) ≡ ((subst τ M) • (σ ⨟ τ)) {B}
sub-dist {Γ}{Δ}{Σ}{A}{B}{σ}{τ}{M} = extensionality λ x → lemma {x = x}
  where
  lemma : ∀ {x : Γ , A ∋ B} → ((M • σ) ⨟ τ) x ≡ ((subst τ M) • (σ ⨟ τ)) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

```
sub-app : ∀{Γ Δ} {σ : Subst Γ Δ} {L : Γ ⊢ ⋆}{M : Γ ⊢ ⋆}
        → ⟪ σ ⟫ (L · M) ≡ (⟪ σ ⟫ L) · (⟪ σ ⟫ M)
sub-app = refl
```

Interlude: congruences

In this section we establish congruence rules for the σ algebra operators `•` and `⨟` and for `subst` and its helper functions `ext`, `rename`, `exts`, and `subst-zero`. These congruence rules help with the equational reasoning in the later sections of this chapter.

[JGS: I would have liked to prove all of these via cong and cong₂, but I have not yet found a way to make that work. It seems that various implicit parameters get in the way.]

```
cong-ext : ∀{Γ Δ}{ρ ρ′ : Rename Γ Δ}{B}
  → (∀{A} → ρ ≡ ρ′ {A})
    ---------------------------------
  → ∀{A} → ext ρ {B = B} ≡ ext ρ′ {A}
cong-ext{Γ}{Δ}{ρ}{ρ′}{B} rr {A} = extensionality λ x → lemma {x}
  where
  lemma : ∀{x : Γ , B ∋ A} → ext ρ x ≡ ext ρ′ x
  lemma {Z} = refl
  lemma {S y} = cong S_ (cong-app rr y)
```

```
cong-rename : ∀{Γ Δ}{ρ ρ′ : Rename Γ Δ}{B}{M : Γ ⊢ B}
  → (∀{A} → ρ ≡ ρ′ {A})
    ------------------------
  → rename ρ M ≡ rename ρ′ M
cong-rename {M = ` x} rr = cong `_ (cong-app rr x)
cong-rename {ρ = ρ} {ρ′ = ρ′} {M = ƛ N} rr =
  cong ƛ_ (cong-rename {ρ = ext ρ}{ρ′ = ext ρ′}{M = N} (cong-ext rr))
cong-rename {M = L · M} rr =
  cong₂ _·_ (cong-rename rr) (cong-rename rr)
```

```
cong-exts : ∀{Γ Δ}{σ σ′ : Subst Γ Δ}{B}
  → (∀{A} → σ ≡ σ′ {A})
    ---------------------------------
  → ∀{A} → exts σ {B = B} ≡ exts σ′ {A}
cong-exts{Γ}{Δ}{σ}{σ′}{B} ss {A} = extensionality λ x → lemma {x}
```

```
  where
  lemma : ∀{x} → exts σ x ≡ exts σ′ x
  lemma {Z} = refl
  lemma {S x} = cong (rename S_) (cong-app (ss {A}) x)
```

```
cong-sub : ∀{Γ Δ}{σ σ′ : Subst Γ Δ}{A}{M M′ : Γ ⊢ A}
            → (∀{A} → σ ≡ σ′ {A}) → M ≡ M′
              ------------------------------
            → subst σ M ≡ subst σ′ M′
cong-sub {Γ} {Δ} {σ} {σ′} {A} {` x} ss refl = cong-app ss x
cong-sub {Γ} {Δ} {σ} {σ′} {A} {ƛ M} ss refl =
  cong ƛ_ (cong-sub {σ = exts σ}{σ′ = exts σ′} {M = M} (cong-exts ss) refl)
cong-sub {Γ} {Δ} {σ} {σ′} {A} {L · M} ss refl =
  cong₂ _·_ (cong-sub {M = L} ss refl) (cong-sub {M = M} ss refl)
```

```
cong-sub-zero : ∀{Γ}{B : Type}{M M′ : Γ ⊢ B}
  → M ≡ M′
    ---------------------------------------
  → ∀{A} → subst-zero M ≡ (subst-zero M′) {A}
cong-sub-zero {Γ}{B}{M}{M′} mm' {A} =
  extensionality λ x → cong (λ z → subst-zero z x) mm'
```

```
cong-cons : ∀{Γ Δ}{A}{M N : Δ ⊢ A}{σ τ : Subst Γ Δ}
  → M ≡ N → (∀{A} → σ {A} ≡ τ {A})
    -------------------------------
  → ∀{A} → (M • σ) {A} ≡ (N • τ) {A}
cong-cons{Γ}{Δ}{A}{M}{N}{σ}{τ} refl st {A′} = extensionality lemma
  where
  lemma : (x : Γ , A ∋ A′) → (M • σ) x ≡ (M • τ) x
  lemma Z = refl
  lemma (S x) = cong-app st x
```

```
cong-seq : ∀{Γ Δ Σ}{σ σ′ : Subst Γ Δ}{τ τ′ : Subst Δ Σ}
  → (∀{A} → σ {A} ≡ σ′ {A}) → (∀{A} → τ {A} ≡ τ′ {A})
  → ∀{A} → (σ ⨟ τ) {A} ≡ (σ′ ⨟ τ′) {A}
cong-seq {Γ}{Δ}{Σ}{σ}{σ′}{τ}{τ′} ss' tt' {A} = extensionality lemma
  where
  lemma : (x : Γ ∋ A) → (σ ⨟ τ) x ≡ (σ′ ⨟ τ′) x
  lemma x =
    begin
      (σ ⨟ τ) x
    ≡⟨⟩
      subst τ (σ x)
    ≡⟨ cong (subst τ) (cong-app ss' x) ⟩
      subst τ (σ′ x)
    ≡⟨ cong-sub{M = σ′ x} tt' refl ⟩
      subst τ′ (σ′ x)
    ≡⟨⟩
      (σ′ ⨟ τ′) x
    ∎
```

Relating `rename`, `exts`, `ext`, and `subst-zero` to the σ algebra

In this section we establish equations that relate `subst` and its helper functions (`rename`, `exts`, `ext`, and `subst-zero`) to terms in the σ algebra.

The first equation we prove is

```
rename ρ M ≡ ⟪ ren ρ ⟫ M                (rename-subst-ren)
```

Because `subst` uses the `exts` function, we need the following lemma which says that `exts` and `ext` do the same thing except that `ext` works on renamings and `exts` works on substitutions.

```
ren-ext : ∀ {Γ Δ}{B C : Type} {ρ : Rename Γ Δ}
        → ren (ext ρ {B = B}) ≡ exts (ren ρ) {C}
ren-ext {Γ}{Δ}{B}{C}{ρ} = extensionality λ x → lemma {x = x}
  where
  lemma : ∀ {x : Γ , B ∋ C} → (ren (ext ρ)) x ≡ exts (ren ρ) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

With this lemma in hand, the proof is a straightforward induction on the term `M`.

```
rename-subst-ren : ∀ {Γ Δ}{A} {ρ : Rename Γ Δ}{M : Γ ⊢ A}
                 → rename ρ M ≡ ⟪ ren ρ ⟫ M
rename-subst-ren {M = ` x} = refl
rename-subst-ren {ρ = ρ}{M = ƛ N} =
  begin
     rename ρ (ƛ N)
    ≡⟨⟩
     ƛ rename (ext ρ) N
    ≡⟨ cong ƛ_ (rename-subst-ren {ρ = ext ρ}{M = N}) ⟩
     ƛ ⟪ ren (ext ρ) ⟫ N
    ≡⟨ cong ƛ_ (cong-sub {M = N} ren-ext refl) ⟩
     ƛ ⟪ exts (ren ρ) ⟫ N
    ≡⟨⟩
     ⟪ ren ρ ⟫ (ƛ N)
  ∎
rename-subst-ren {M = L · M} = cong₂ _·_ rename-subst-ren rename-subst-ren
```

The substitution `ren S_` is equivalent to `↑`.

```
ren-shift : ∀{Γ}{A}{B}
          → ren S_ ≡ ↑ {A = B} {A}
ren-shift {Γ}{A}{B} = extensionality λ x → lemma {x = x}
  where
  lemma : ∀ {x : Γ ∋ A} → ren (S_ {B = B}) x ≡ ↑ {A = B} x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

The substitution `rename S_ M` is equivalent to shifting: `⟪ ↑ ⟫ M`.

```
rename-shift : ∀{Γ} {A} {B} {M : Γ ⊢ A}
             → rename (S_ {B = B}) M ≡ ⟪ ↑ ⟫ M
rename-shift{Γ}{A}{B}{M} =
  begin
```

```
    rename S_ M
  ≡⟨ rename-subst-ren ⟩
    ⟪ ren S_ ⟫ M
  ≡⟨ cong-sub{M = M} ren-shift refl ⟩
    ⟪ ↑ ⟫ M
  ∎
```

Next we prove the equation `exts-cons-shift`, which states that `exts` is equivalent to cons'ing Z onto the sequence formed by applying `σ` and then shifting. The proof is by case analysis on the variable `x`, using `rename-subst-ren` for when `x = S y`.

```
exts-cons-shift : ∀{Γ Δ} {A B} {σ : Subst Γ Δ}
                → exts σ {A} {B} ≡ (` Z • (σ ⨟ ↑))
exts-cons-shift = extensionality λ x → lemma{x = x}
  where
  lemma : ∀{Γ Δ} {A B} {σ : Subst Γ Δ} {x : Γ , B ∋ A}
                → exts σ x ≡ (` Z • (σ ⨟ ↑)) x
  lemma {x = Z} = refl
  lemma {x = S y} = rename-subst-ren
```

As a corollary, we have a similar correspondence for `ren (ext ρ)`.

```
ext-cons-Z-shift : ∀{Γ Δ} {ρ : Rename Γ Δ}{A}{B}
                → ren (ext ρ {B = B}) ≡ (` Z • (ren ρ ⨟ ↑)) {A}
ext-cons-Z-shift {Γ}{Δ}{ρ}{A}{B} =
  begin
    ren (ext ρ)
  ≡⟨ ren-ext ⟩
    exts (ren ρ)
  ≡⟨ exts-cons-shift{σ = ren ρ} ⟩
    ((` Z) • (ren ρ ⨟ ↑))
  ∎
```

Finally, the `subst-zero M` substitution is equivalent to cons'ing `M` onto the identity substitution.

```
subst-Z-cons-ids : ∀{Γ}{A B : Type}{M : Γ ⊢ B}
                → subst-zero M ≡ (M • ids) {A}
subst-Z-cons-ids = extensionality λ x → lemma {x = x}
  where
  lemma : ∀{Γ}{A B : Type}{M : Γ ⊢ B}{x : Γ , B ∋ A}
                   → subst-zero M x ≡ (M • ids) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

Proofs of sub-abs, sub-id, and rename-id

The equation `sub-abs` follows immediately from the equation `exts-cons-shift`.

```
sub-abs : ∀{Γ Δ} {σ : Subst Γ Δ} {N : Γ , ⋆ ⊢ ⋆}
        → ⟪ σ ⟫ (ƛ N) ≡ ƛ ⟪ (` Z) • (σ ⨟ ↑) ⟫ N
sub-abs {σ = σ}{N = N} =
  begin
    ⟪ σ ⟫ (ƛ N)
```

```
  ≡⟨⟩
    ƛ ⟪ exts σ ⟫ N
  ≡⟨ cong ƛ_ (cong-sub{M = N} exts-cons-shift refl) ⟩
    ƛ ⟪ (` Z) • (σ ⨟ ↑) ⟫ N
  ∎
```

The proof of `sub-id` requires the following lemma which says that extending the identity substitution produces the identity substitution.

```
exts-ids : ∀{Γ}{A B}
         → exts ids ≡ ids {Γ , B} {A}
exts-ids {Γ}{A}{B} = extensionality lemma
  where lemma : (x : Γ , B ∋ A) → exts ids x ≡ ids x
        lemma Z = refl
        lemma (S x) = refl
```

The proof of `⟪ ids ⟫ M ≡ M` now follows easily by induction on `M`, using `exts-ids` in the case for `M ≡ ƛ N`.

```
sub-id : ∀{Γ} {A} {M : Γ ⊢ A}
         → ⟪ ids ⟫ M ≡ M
sub-id {M = ` x} = refl
sub-id {M = ƛ N} =
  begin
    ⟪ ids ⟫ (ƛ N)
  ≡⟨⟩
    ƛ ⟪ exts ids ⟫ N
  ≡⟨ cong ƛ_ (cong-sub{M = N} exts-ids refl) ⟩
    ƛ ⟪ ids ⟫ N
  ≡⟨ cong ƛ_ sub-id ⟩
    ƛ N
  ∎
sub-id {M = L · M} = cong₂ _·_ sub-id sub-id
```

The `rename-id` equation is a corollary is `sub-id`.

```
rename-id : ∀ {Γ}{A} {M : Γ ⊢ A}
  → rename (λ {A} x → x) M ≡ M
rename-id {M = M} =
  begin
    rename (λ {A} x → x) M
  ≡⟨ rename-subst-ren ⟩
    ⟪ ren (λ {A} x → x) ⟫ M
  ≡⟨⟩
    ⟪ ids ⟫ M
  ≡⟨ sub-id ⟩
    M
  ∎
```

Proof of sub-idR

The proof of `sub-idR` follows directly from `sub-id`.

```
sub-idR : ∀{Γ Δ} {σ : Subst Γ Δ} {A}
  → (σ ⨟ ids) ≡ σ {A}
sub-idR {Γ}{σ = σ}{A} =
         begin
           σ ⨟ ids
         ≡⟨⟩
           ⟪ ids ⟫ ∘ σ
         ≡⟨ extensionality (λ x → sub-id) ⟩
           σ
         ∎
```

Proof of sub-sub

The `sub-sub` equation states that sequenced substitutions `σ ⨟ τ` are equivalent to first applying `σ` then applying `τ`.

```
⟪ τ ⟫ ⟪ σ ⟫ M  ≡ ⟪ σ ⨟ τ ⟫ M
```

The proof requires several lemmas. First, we need to prove the specialization for renaming.

```
rename ρ (rename ρ′ M) ≡ rename (ρ ∘ ρ′) M
```

This in turn requires the following lemma about `ext`.

```
compose-ext : ∀{Γ Δ Σ}{ρ : Rename Δ Σ} {ρ′ : Rename Γ Δ} {A B}
            → ((ext ρ) ∘ (ext ρ′)) ≡ ext (ρ ∘ ρ′) {B} {A}
compose-ext = extensionality λ x → lemma {x = x}
  where
  lemma : ∀{Γ Δ Σ}{ρ : Rename Δ Σ} {ρ′ : Rename Γ Δ} {A B} {x : Γ , B ∋ A}
            → ((ext ρ) ∘ (ext ρ′)) x ≡ ext (ρ ∘ ρ′) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

To prove that composing renamings is equivalent to applying one after the other using `rename`, we proceed by induction on the term `M`, using the `compose-ext` lemma in the case for `M ≡ ƛ N`.

```
compose-rename : ∀{Γ Δ Σ}{A}{M : Γ ⊢ A}{ρ : Rename Δ Σ}{ρ′ : Rename Γ Δ}
  → rename ρ (rename ρ′ M) ≡ rename (ρ ∘ ρ′) M
compose-rename {M = ` x} = refl
compose-rename {Γ}{Δ}{Σ}{A}{ƛ N}{ρ}{ρ′} = cong ƛ_ G
  where
  G : rename (ext ρ) (rename (ext ρ′) N) ≡ rename (ext (ρ ∘ ρ′)) N
  G =
     begin
       rename (ext ρ) (rename (ext ρ′) N)
     ≡⟨ compose-rename{ρ = ext ρ}{ρ′ = ext ρ′} ⟩
       rename ((ext ρ) ∘ (ext ρ′)) N
     ≡⟨ cong-rename compose-ext ⟩
       rename (ext (ρ ∘ ρ′)) N
     ∎
compose-rename {M = L · M} = cong₂ _·_ compose-rename compose-rename
```

The next lemma states that if a renaming and substitution commute on variables, then they also commute on terms. We explain the proof in detail below.

```
commute-subst-rename : ∀{Γ Δ}{M : Γ ⊢ ★}{σ : Subst Γ Δ}
                       {ρ : ∀{Γ} → Rename Γ (Γ , ★)}
   → (∀{x : Γ ∋ ★} → exts σ {B = ★} (ρ x) ≡ rename ρ (σ x))
   → subst (exts σ {B = ★}) (rename ρ M) ≡ rename ρ (subst σ M)
commute-subst-rename {M = ` x} r = r
commute-subst-rename{Γ}{Δ}{ƛ N}{σ}{ρ} r =
  cong ƛ_ (commute-subst-rename{Γ , ★}{Δ , ★}{N}
           {exts σ}{ρ = ρ′} (λ {x} → H {x}))
  where
  ρ′ : ∀ {Γ} → Rename Γ (Γ , ★)
  ρ′ {∅} = λ ()
  ρ′ {Γ , ★} = ext ρ

  H : {x : Γ , ★ ∋ ★} → exts (exts σ) (ext ρ x) ≡ rename (ext ρ) (exts σ x)
  H {Z} = refl
  H {S y} =
    begin
      exts (exts σ) (ext ρ (S y))
    ≡⟨⟩
      rename S_ (exts σ (ρ y))
    ≡⟨ cong (rename S_) r ⟩
      rename S_ (rename ρ (σ y))
    ≡⟨ compose-rename ⟩
      rename (S_ ∘ ρ) (σ y)
    ≡⟨ cong-rename refl ⟩
      rename ((ext ρ) ∘ S_) (σ y)
    ≡⟨ sym compose-rename ⟩
      rename (ext ρ) (rename S_ (σ y))
    ≡⟨⟩
      rename (ext ρ) (exts σ (S y))
    ∎
commute-subst-rename {M = L · M}{ρ = ρ} r =
  cong₂ _·_ (commute-subst-rename{M = L}{ρ = ρ} r)
            (commute-subst-rename{M = M}{ρ = ρ} r)
```

The proof is by induction on the term `M`.

- If `M` is a variable, then we use the premise to conclude.
- If `M ≡ ƛ N`, we conclude using the induction hypothesis for `N`. However, to use the induction hypothesis, we must show that

  ```
  exts (exts σ) (ext ρ x) ≡ rename (ext ρ) (exts σ x)
  ```

  We prove this equation by cases on x.

  - If `x = Z`, the two sides are equal by definition.
  - If `x = S y`, we obtain the goal by the following equational reasoning.

    ```
      exts (exts σ) (ext ρ (S y))
    ≡ rename S_ (exts σ (ρ y))
    ≡ rename S_ (rename S_ (σ (ρ y)      (by the premise)
    ≡ rename (ext ρ) (exts σ (S y))      (by compose-rename)
    ≡ rename ((ext ρ) ∘ S_) (σ y)
    ≡ rename (ext ρ) (rename S_ (σ y))   (by compose-rename)
    ≡ rename (ext ρ) (exts σ (S y))
    ```

- If `M` is an application, we obtain the goal using the induction hypothesis for each subterm.

The last lemma needed to prove `sub-sub` states that the `exts` function distributes with sequencing. It is a corollary of `commute-subst-rename` as described below. (It would have been nicer to prove this directly by equational reasoning in the σ algebra, but that would require the `sub-assoc` equation, whose proof depends on `sub-sub`, which in turn depends on this lemma.)

```
exts-seq : ∀{Γ Δ Δ′} {σ₁ : Subst Γ Δ} {σ₂ : Subst Δ Δ′}
         → ∀ {A} → (exts σ₁ ⨟ exts σ₂) {A} ≡ exts (σ₁ ⨟ σ₂)
exts-seq = extensionality λ x → lemma {x = x}
  where
  lemma : ∀{Γ Δ Δ′}{A}{x : Γ , ⋆ ∋ A} {σ₁ : Subst Γ Δ}{σ₂ : Subst Δ Δ′}
    → (exts σ₁ ⨟ exts σ₂) x ≡ exts (σ₁ ⨟ σ₂) x
  lemma {x = Z} = refl
  lemma {A = ⋆}{x = S x}{σ₁}{σ₂} =
    begin
      (exts σ₁ ⨟ exts σ₂) (S x)
    ≡⟨⟩
      ⟪ exts σ₂ ⟫ (rename S_ (σ₁ x))
    ≡⟨ commute-subst-rename{M = σ₁ x}{σ = σ₂}{ρ = S_} refl ⟩
      rename S_ (⟪ σ₂ ⟫ (σ₁ x))
    ≡⟨⟩
      rename S_ ((σ₁ ⨟ σ₂) x)
    ∎
```

The proof proceed by cases on `x`.

- If `x = Z`, the two sides are equal by the definition of `exts` and sequencing.
- If `x = S x`, we unfold the first use of `exts` and sequencing, then apply the lemma `commute-subst-rename`. We conclude by the definition of sequencing.

Now we come to the proof of `sub-sub`, which we explain below.

```
sub-sub : ∀{Γ Δ Σ}{A}{M : Γ ⊢ A} {σ₁ : Subst Γ Δ}{σ₂ : Subst Δ Σ}
            → ⟪ σ₂ ⟫ (⟪ σ₁ ⟫ M) ≡ ⟪ σ₁ ⨟ σ₂ ⟫ M
sub-sub {M = ` x} = refl
sub-sub {Γ}{Δ}{Σ}{A}{ƛ N}{σ₁}{σ₂} =
  begin
    ⟪ σ₂ ⟫ (⟪ σ₁ ⟫ (ƛ N))
  ≡⟨⟩
    ƛ ⟪ exts σ₂ ⟫ (⟪ exts σ₁ ⟫ N)
  ≡⟨ cong ƛ_ (sub-sub{M = N}{σ₁ = exts σ₁}{σ₂ = exts σ₂}) ⟩
    ƛ ⟪ exts σ₁ ⨟ exts σ₂ ⟫ N
  ≡⟨ cong ƛ_ (cong-sub{M = N} (λ {A} → exts-seq) refl) ⟩
    ƛ ⟪ exts (σ₁ ⨟ σ₂) ⟫ N
  ∎
sub-sub {M = L · M} = cong₂ _·_ (sub-sub{M = L}) (sub-sub{M = M})
```

We proceed by induction on the term `M`.

- If `M = x`, then both sides are equal to `σ₂ (σ₁ x)`.
- If `M = ƛ N`, we first use the induction hypothesis to show that

  ```
  ƛ ⟪ exts σ₂ ⟫ (⟪ exts σ₁ ⟫ N) ≡ ƛ ⟪ exts σ₁ ⨟ exts σ₂ ⟫ N
  ```

  and then use the lemma `exts-seq` to show

```
    ↚ ⟪ exts σ₁ ⨟ exts σ₂ ⟫ N ≡ ↚ ⟪ exts ( σ₁ ⨟ σ₂) ⟫ N
```

- If `M` is an application, we use the induction hypothesis for both subterms.

The following corollary of `sub-sub` specializes the first substitution to a renaming.

```
rename-subst : ∀{Γ Δ Δ′}{M : Γ ⊢ ⋆}{ρ : Rename Γ Δ}{σ : Subst Δ Δ′}
             → ⟪ σ ⟫ (rename ρ M) ≡ ⟪ σ ∘ ρ ⟫ M
rename-subst {Γ}{Δ}{Δ′}{M}{ρ}{σ} =
  begin
    ⟪ σ ⟫ (rename ρ M)
  ≡⟨ cong ⟪ σ ⟫ (rename-subst-ren{M = M}) ⟩
    ⟪ σ ⟫ (⟪ ren ρ ⟫ M)
  ≡⟨ sub-sub{M = M} ⟩
    ⟪ ren ρ ⨟ σ ⟫ M
  ≡⟨⟩
    ⟪ σ ∘ ρ ⟫ M
  ∎
```

Proof of sub-assoc

The proof of `sub-assoc` follows directly from `sub-sub` and the definition of sequencing.

```
sub-assoc : ∀{Γ Δ Σ Ψ : Context} {σ : Subst Γ Δ} {τ : Subst Δ Σ}
              {θ : Subst Σ Ψ}
          → ∀{A} → (σ ⨟ τ) ⨟ θ ≡ (σ ⨟ τ ⨟ θ) {A}
sub-assoc {Γ}{Δ}{Σ}{Ψ}{σ}{τ}{θ}{A} = extensionality λ x → lemma{x = x}
  where
  lemma : ∀ {x : Γ ∋ A} → ((σ ⨟ τ) ⨟ θ) x ≡ (σ ⨟ τ ⨟ θ) x
  lemma {x} =
    begin
      ((σ ⨟ τ) ⨟ θ) x
    ≡⟨⟩
      ⟪ θ ⟫ (⟪ τ ⟫ (σ x))
    ≡⟨ sub-sub{M = σ x} ⟩
      ⟪ τ ⨟ θ ⟫ (σ x)
    ≡⟨⟩
      (σ ⨟ τ ⨟ θ) x
    ∎
```

Proof of subst-zero-exts-cons

The last equation we needed to prove `subst-zero-exts-cons` was `sub-assoc`, so we can now go ahead with its proof. We simply apply the equations for `exts` and `subst-zero` and then apply the σ algebra equation to arrive at the normal form `M • σ`.

```
subst-zero-exts-cons : ∀{Γ Δ}{σ : Subst Γ Δ}{B}{M : Δ ⊢ B}{A}
                     → exts σ ⨟ subst-zero M ≡ (M • σ) {A}
subst-zero-exts-cons {Γ}{Δ}{σ}{B}{M}{A} =
  begin
```

```
    exts σ ⨟ subst-zero M
  ≡⟨ cong-seq exts-cons-shift subst-Z-cons-ids ⟩
    (` Z • (σ ⨟ ↑)) ⨟ (M • ids)
  ≡⟨ sub-dist ⟩
    (⟪ M • ids ⟫ (` Z)) • ((σ ⨟ ↑) ⨟ (M • ids))
  ≡⟨ cong-cons (sub-head{σ = ids}) refl ⟩
    M • ((σ ⨟ ↑) ⨟ (M • ids))
  ≡⟨ cong-cons refl (sub-assoc{σ = σ}) ⟩
    M • (σ ⨟ (↑ ⨟ (M • ids)))
  ≡⟨ cong-cons refl (cong-seq{σ = σ} refl (sub-tail{M = M}{σ = ids})) ⟩
    M • (σ ⨟ ids)
  ≡⟨ cong-cons refl (sub-idR{σ = σ}) ⟩
    M • σ
  ∎
```

Proof of the substitution lemma

We first prove the generalized form of the substitution lemma, showing that a substitution `σ` commutes with the substitution of `M` into `N`.

```
⟪ exts σ ⟫ N [ ⟪ σ ⟫ M ] ≡ ⟪ σ ⟫ (N [ M ])
```

This proof is where the σ algebra pays off. The proof is by direct equational reasoning. Starting with the left-hand side, we apply σ algebra equations, oriented left-to-right, until we arrive at the normal form

```
⟪ ⟪ σ ⟫ M • σ ⟫ N
```

We then do the same with the right-hand side, arriving at the same normal form.

```
subst-commute : ∀{Γ Δ}{N : Γ , ⋆ ⊢ ⋆}{M : Γ ⊢ ⋆}{σ : Subst Γ Δ }
  → ⟪ exts σ ⟫ N [ ⟪ σ ⟫ M ] ≡ ⟪ σ ⟫ (N [ M ])
subst-commute {Γ}{Δ}{N}{M}{σ} =
  begin
    ⟪ exts σ ⟫ N [ ⟪ σ ⟫ M ]
  ≡⟨⟩
    ⟪ subst-zero (⟪ σ ⟫ M) ⟫ (⟪ exts σ ⟫ N)
  ≡⟨ cong-sub {M = ⟪ exts σ ⟫ N} subst-Z-cons-ids refl ⟩
    ⟪ ⟪ σ ⟫ M • ids ⟫ (⟪ exts σ ⟫ N)
  ≡⟨ sub-sub {M = N} ⟩
    ⟪ (exts σ) ⨟ ((⟪ σ ⟫ M) • ids) ⟫ N
  ≡⟨ cong-sub {M = N} (cong-seq exts-cons-shift refl) refl ⟩
    ⟪ (` Z • (σ ⨟ ↑)) ⨟ (⟪ σ ⟫ M • ids) ⟫ N
  ≡⟨ cong-sub {M = N} (sub-dist {M = ` Z}) refl ⟩
    ⟪ ⟪ ⟪ σ ⟫ M • ids ⟫ (` Z) • ((σ ⨟ ↑) ⨟ (⟪ σ ⟫ M • ids)) ⟫ N
  ≡⟨⟩
    ⟪ ⟪ σ ⟫ M • ((σ ⨟ ↑) ⨟ (⟪ σ ⟫ M • ids)) ⟫ N
  ≡⟨ cong-sub{M = N} (cong-cons refl (sub-assoc{σ = σ})) refl ⟩
    ⟪ ⟪ σ ⟫ M • (σ ⨟ ↑ ⨟ ⟪ σ ⟫ M • ids) ⟫ N
  ≡⟨ cong-sub{M = N} refl refl ⟩
    ⟪ ⟪ σ ⟫ M • (σ ⨟ ids) ⟫ N
  ≡⟨ cong-sub{M = N} (cong-cons refl (sub-idR{σ = σ})) refl ⟩
    ⟪ ⟪ σ ⟫ M • σ ⟫ N
  ≡⟨ cong-sub{M = N} (cong-cons refl (sub-idL{σ = σ})) refl ⟩
    ⟪ ⟪ σ ⟫ M • (ids ⨟ σ) ⟫ N
```

```
  ≡⟨ cong-sub{M = N} (sym sub-dist) refl ⟩
    ⟪ M • ids ⨟ σ ⟫ N
  ≡⟨ sym (sub-sub{M = N}) ⟩
    ⟪ σ ⟫ (⟪ M • ids ⟫ N)
  ≡⟨ cong ⟪ σ ⟫ (sym (cong-sub{M = N} subst-Z-cons-ids refl)) ⟩
    ⟪ σ ⟫ (N [ M ])
  ∎
```

A corollary of `subst-commute` is that `rename` also commutes with substitution. In the proof below, we first exchange `rename ρ` for the substitution `⟪ ren ρ ⟫`, and apply `subst-commute`, and then convert back to `rename ρ`.

```
rename-subst-commute : ∀{Γ Δ}{N : Γ , ⋆ ⊢ ⋆}{M : Γ ⊢ ⋆}{ρ : Rename Γ Δ }
  → (rename (ext ρ) N) [ rename ρ M ] ≡ rename ρ (N [ M ])
rename-subst-commute {Γ}{Δ}{N}{M}{ρ} =
  begin
    (rename (ext ρ) N) [ rename ρ M ]
  ≡⟨ cong-sub (cong-sub-zero (rename-subst-ren{M = M}))
              (rename-subst-ren{M = N}) ⟩
    (⟪ ren (ext ρ) ⟫ N) [ ⟪ ren ρ ⟫ M ]
  ≡⟨ cong-sub refl (cong-sub{M = N} ren-ext refl) ⟩
    (⟪ exts (ren ρ) ⟫ N) [ ⟪ ren ρ ⟫ M ]
  ≡⟨ subst-commute{N = N} ⟩
    subst (ren ρ) (N [ M ])
  ≡⟨ sym (rename-subst-ren) ⟩
    rename ρ (N [ M ])
  ∎
```

To present the substitution lemma, we introduce the following notation for substituting a term `M` for index 1 within term `N`.

```
_[_] : ∀ {Γ A B C}
       → Γ , B , C ⊢ A
       → Γ ⊢ B
         ---------
       → Γ , C ⊢ A
_[_] {Γ} {A} {B} {C} N M =
  subst {Γ , B , C} {Γ , C} (exts (subst-zero M)) {A} N
```

The substitution lemma is stated as follows and proved as a corollary of the `subst-commute` lemma.

```
substitution : ∀{Γ}{M : Γ , ⋆ , ⋆ ⊢ ⋆}{N : Γ , ⋆ ⊢ ⋆}{L : Γ ⊢ ⋆}
  → (M [ N ]) [ L ] ≡ (M [ L ]) [ (N [ L ]) ]
substitution{M = M}{N = N}{L = L} =
  sym (subst-commute{N = M}{M = N}{σ = subst-zero L})
```

Notes

Most of the properties and proofs in this file are based on the paper *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitution* by Schafer, Tebbi, and Smolka (ITP 2015). That paper, in turn, is based on the paper of Abadi, Cardelli, Curien, and Levy (1991) that defines the σ algebra.

Unicode

This chapter uses the following unicode:

```
⟪  U+27EA  MATHEMATICAL LEFT DOUBLE ANGLE BRACKET (\<<)
⟫  U+27EA  MATHEMATICAL RIGHT DOUBLE ANGLE BRACKET (\>>)
↑  U+2191  UPWARDS ARROW (\u)
•  U+2022  BULLET (\bub)
⨟  U+2A1F  Z NOTATION SCHEMA COMPOSITION (C-x 8 RET Z NOTATION SCHEMA COMPOSITION)
〔  U+3014  LEFT TORTOISE SHELL BRACKET (\( option 9 on page 2)
〕  U+3015  RIGHT TORTOISE SHELL BRACKET (\) option 9 on page 2)
```

Part V

Back matter

Appendix B

Acknowledgements

Thank you to:

- The inventors of Agda, for a new playground.
- The authors of Software Foundations, for inspiration.

A special thank you, for inventing ideas on which this book is based, and for hand-holding:

Andreas Abel

Catarina Coquand

Thierry Coquand

David Darais

Per Martin-Löf

Lena Magnusson

Conor McBride

James McKinna

Ulf Norell

For pull requests big and small, and for answering questions on the Agda mailing list:

Marko Dimjašević

Zbigniew Stanasiuk

Reza Gharibi

Yasu Watanabe

Michael Reed

Chad Nester

Fangyi Zhou

Mo Mirza

Juhana Laurinharju

Qais Patankar

Orestis Melkonian

Kenichi Asai

phi16

Pedro Minicz

Jonathan Prieto

Alexandru Brisan

Sebastian Miele

bc²

Murilo Giacometti Rocha

Michel Steuwer

Matthew Healy

Lorenzo Martinico

caryoscelus

Zach Brown

Syed Turab Ali Jafri

Spencer Whitt

Peter Thiemann

Alexandre Moreno

Ingo Blechschmidt

G. Allais

Matthias Gabriel

Isaac Elliott

Liang-Ting Chen

Mike He

Zack Grannan

Nicolas Wu

Slava

Vikraman Choudhury

Stephan Boyer

Amr Sabry

Rodrigo Bernardo

purchan

Nathaniel Carroll

Phil de Joux

N. Raghavendra

Léo Gillot-Lamure

Nils Anders Danielsson

Miëtek Bak

Merlin Göttlinger

Liam O’Connor

James Wood

Kenneth MacKenzie

Stefan Kranich

koo5

Kartik Singhal

John Maraist

Hugo Gualandi

Gan Shen

Georgi Lyubenov

Gergő Érdi

Roman Kireev

David Janin

Deniz Alp

April Gonçalves

citrusmunch

Chike Abuah

Ben Darwin

Anish Tondwalkar

Adam Sandberg Eriksson

Ulf Norell

Torsten Grust

Oling Cat

Gagan Devagiri

Dee Yeum

András Kovács

[Your name goes here]

For contributions to the answers repository:

William Cook

David Banas

There is a private repository of answers to selected questions on github. Please contact Philip Wadler if you would like to access it.

For support:

- EPSRC Programme Grant EP/K034413/1
- NSF Grant No. 1814460
- Foundation Sciences Mathematiques de Paris (FSMP) Distinguised Professor Fellowship

Appendix C

Fonts

```
module plfa.backmatter.Fonts where
```

Preferably, all vertical bars should line up.

```
--------------------------|
abcdefghijklmnopqrstuvwxyz|
ABCDEFGHIJKLMNOPQRSTUVWXYZ|
a b c d e f g h i j k l m n o p  r s t u v w x y z|
A B  D E  G H I J K L M N O P  R  T U V W     |
a     e    h i j k l m n o p  r s t u   x    |
--------------------------|
----------|
0123456789|
0 1 2 3 4 5 6 7 8 9|
          |
0 1 2 3 4 5 6 7 8 9|
----------|
-------------------------|
αβγδεζηθικλμνξοπρστυφχψω|
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ|
-------------------------|
----|
≠≠≠≠|
ηηµµ|
ΓΓΔΔ|
ΣΣΠΠ|
λλλλ|
ǂǂǂǂ|
····|
××××|
ℓℓℓℓ|
≡≡≡≡|
¬¬¬¬|
≤≤≥≥|
øøøø|
————|
††‡‡|
^^^^|
'''"|
``~~|
ⱭⱭↃↃ|
∧∧∨∨|
```

```
⊗⊗⊗⊗|
⊔⊔⊔⊔|
cᶜbᵇ|
lˡrʳ|
−−++|
𝒩𝒩𝒩𝒩|
∀∀∃∃|
′′″″|
∘∘∘∘|
≠≠≃≃|
≲≲≳≳|
≐≐≐≐|
⋅⋅⋅⋅|
⟨⟨⟩⟩|
⌊⌊⌋⌋|
⌈⌈⌉⌉|
↑↑↓↓|
⇔⇔⇔⇔|
⇾⇾⇒⇒|
⇽⇽⇐⇐|
⇻⇻⇺⇺|
∈∈∋∋|
⊢⊢⊣⊣|
⊥⊥⊤⊤|
::::::::|
▮▮▮▮|
⠿⠿⠿⠿|
||||||||
★★★★|
∄∄∉∉|
⸱⸱⸱⸱|
⎣⎣⎦⎦|
⟦⟦⟧⟧|
----|
```

In the book we use the em-dash to make big arrows.

```
----|
⟶⟶|
⟵⟵|
⇜⇜|
⇝⇝|
----|
```

Here are some characters that are often not monospaced.

```
----|
😇😇|
😈😈|
′′′′|
‴‴|
----|
------------|
------------|
----------|
𝔸𝔹𝔻𝔼𝔽𝔾𝕀𝕁𝕂𝕃𝕄ℕ𝕆𝕊|
𝖆𝖇𝖈𝖉𝖊𝖋𝖌𝖍𝖎𝖏|
𝑎𝑏𝑐𝑑𝑒𝑓𝑔𝑖𝑗𝑘|
𝓔𝓕|
```

```
----------|
```