Philip Wadler, Wen Kokke, and Jeremy G. Siek

PROGRAMMING LANGUAGE FOUNDATIONS

IN

Agda

2021

目录

Dedication: 题献

致 Wanda

我生命中的至爱

咚 咚 咚

Philip 致 Wanda

我生命中的至爱

咚 咚 咚

...

Preface: 前言

逻辑与计算之间最深刻的联系是一种双关。「命题即类型（Propositions as Types）」的学说断言，形式化的结构可以按两种方式看待：可以看做逻辑中的命题，也可以看做计算中的类型。此外，相关的结构可以看做命题的证明或者其相应类型的程序。更进一步来说，证明的化简与程序的求值对应。

与此相应，本书的名字也有两种含义。它可以看做「**编程语言**的基础」，也可以看做「编程的**语言基础**」。我们用 Agda 证明助理编写的规范（Specification）同时描述了编程语言以及该语言编写的程序自身。

本书面向本科最后一年的学生、一年级的研究生或博士生。 本书以简单类型 λ-演算（Simply-Typed Lambda Calculus，简称 STLC）作为核心示例，旨在教授编程语言的操作语义基础。全书以 Agda 文学脚本（Literal Script）的形式写成。使用证明助理可以让开发过程变得更加具体而清晰易懂，还可以给予学生即时反馈，帮助学生发现理解有误的地方并及时纠正。

【译注：文学编程（Literal Programming）用带有代码的自然语言文章来进行编程，此概念由高德纳（Donald Knuth）提出，详情可参考维基百科。】

本书分为两册。第一分册为逻辑基础，发展了所需的形式化工具。第二分册为编程语言基础，介绍了操作语义的基本方法。

个人评论

从 2013 年开始，我在爱丁堡大学为四年制本科生和研究生教授编程语言的类型和语义的课程。该课程的早期版本基于 Benjamin Pierce 的著作 TAPL。我的版本则基于 Pierce 的后续教材《软件基础》（英文版 Software Foundations），此书为 Pierce 与他人合著，基于 Coq 编写。正如 Pierce 在 ICFP 的主题演讲 Lambda, The Ultimate TA 中所言，我也相信基于证明助理的课程会对学习有所帮助。

然而有了五年的教学经验后，我发现 Coq 并不是最好的授课载体。对于学习编程语言理论基础而言，我们花费了太多课程去专门学习证明推导的策略（Tactic）。每个概念都需要学习两遍：例如，在学过一遍积数据类型（Product Data Type）之后，我们还要再学一遍与之对应的合取（Conjunction）的引入（Introduction）和消除（Elimination）策略。Coq 用来生成归纳假设（Induction Hypothesis）的规则有时看起来很玄学。而 `notation` 构造则允许直观但灵活多变的语法，同一个概念总是有两个名字有时会令人迷惑，例如，`subst N x M` 和 `N [x := M]`。策略的名字时短时长；标准库中的命名约定则非常不一致。**命题即类型**作为证明的基础虽然存在，但却被雪藏了。

我发现自己热衷于用 Agda 重构此课程。在 Agda 中，我们不再需要学习策略了：这里只有依赖类型编程，简单纯粹。我们总是通过构造子来引入，通过模式匹配来消除。归纳不再是谜之独立的概念，它与我们熟悉的递归概念直接对应。混缀语法十分灵活，但每个概念只需要一个名字，例如代换就是 `_[_:=_]`。标准库虽不完美，但它的一致性却很合理。**命题即类型**作为证明的基础则被骄傲地展示了出来。

不过，此前还没有用 Agda 语言描述的编程语言理论教材。虽然 Stump 的 Verified Functional Programming in Agda 涵盖了相关的范围，但比起编程语言理论，却更多关注于依赖类型编程。

本书最初的目标只是简单地改编《**软件基础**》，保持同样的内容，而只是将代码从 Coq 翻译成 Agda。但五年的课堂经验很快让我明白，自己有一些关于如何展示这些材料的想法。有人说除非你**不得不**写书，否则绝对不要写书。而我很快发现这就是一本我不得不写的书。

我很幸运地得到了我的学生 Wen Kokke 的热情帮助。她引导着我站在 Agda 新手的视角写书，还提供了一些十分易用的工具来生成清晰易读的页面。

这里的大部分内容都是在 2018 年上半年的休假期间写的。

—— Philip Wadler，里约热内卢，2018 年 1 月 - 6 月

习题说明

标有**推荐**的习题是爱丁堡大学使用本教材的课程中学生需要做的。

标有**延伸**的习题提供了额外的挑战。很少有学生能全部做完，但至少应该尝试一下。

标记有**实践**的习题供想要更多实践的学生练习。

你在解题时或许需要导入库函数。

请不要在公共空间中发表习题的答案。

GitHub 上有一个发布了部分习题答案的非公开代码仓库。如果你想获得访问，请联系 Philip Wadler。

使用说明

《编程语言基础：Agda 语言描述》的使用方法与《Programming Language Foundations in Agda》一致。

本书可访问 PLFA-zh 在线阅读。

要参与翻译，请先阅读翻译规范。

用户依赖

你可以在线阅读 PLFA，无需安装任何东西。然而，如果你想要交互式编写代码或完成习题，那么就需要几样东西：

- Stack
- Git
- Agda
- Agda 标准库
- PLFA

PLFA 只针对特定的 Agda 和标准库版本进行了测试，相应版本已在前面的徽章中指明。 Agda 和标准库变化得十分迅速，而这些改变经常搞坏 PLFA，因此使用旧版或新版通常会出现问题。

Agda 和 Agda 标准库有多个版本。如果你使用了包管理器（如 Homebrew 或者 Debian apt），Agda 的版本可能不是最新的。除此之外，Agda 仍然在活跃的开发之中，如果你从 GitHub 上安装了开发版本，开发者的新变更也可能让这里的代码出现问题。因此，使用上述版本的 Agda 和 Agda 标准库很重要。

macOS 平台：安装 XCode 命令行工具

在 macOS 平台，你需要安装 XCode 命令行工具。 在大多数 macOS 系统版本上，你可以用下面的命令安装它们：

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

安装 Haskell 工具 Stack

Agda 使用 Haskell 写的，因此我们需要 Haskell 工具 Stack （简称 Stack）来安装 Agda。 Stack 是管理不同版本的 Haskell 编译器和包的管理程序。

- **UNIX 和 macOS** 平台：如果你的包管理程序有 Stack 的包，这是大概是最简单的方法。 比如说 macOS 的 Homebrew 或者 Debian 的 APT 提供了「haskell-stack」包。 不然，你可以按照 Stack 网站上的指示进行安装。 在一般情况下，Stack 将二进制文件安装至 `HOME/.local/bin` 。 请务必保证这个目录在你的 PATH 之内，你可以将下面的内容加入你的 shell 配置中，例如 `HOME/.bash_profile` 中：

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

  最后，请保证你有最新版的 Stack，运行：

  ```
  stack upgrade
  ```

- **Windows** 平台：Stack 网站上提供 Windows 安装包。

安装 Git

如果你没有已经安装 Git，请参阅 Git 下载页面。

用 Stack 安装 Agda

安装 **特定** Agda 版本的最简方式是使用 Stack。你可以从 GitHub 上获取需要的版本，可以通过克隆源码库并切换到合适的分支，也可以直接下载 Zip 包：

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

要安装 Agda，请在其源码目录中运行 Stack ：

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

这一步会消耗很长时间和很多内存来完成。

使用已安装的 GHC

Stack 可以为你安装和管理 Glasgow Haskell 编译器（GHC）。 然而，如果你已经安装了 GHC 并且想让 Stack 使用你系统上安装的 GHC 版本，可以传递 `--system-ghc` 选项并选择对应的 `stack-*.yaml` 文件。例如，若你安装了 GHC 8.2.2，请运行：

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

检查 Agda 是否被正确地安装

如果你愿意，你可以检查 Agda 是否已正确安装。 创建一个名为 `hello.agda` 的文件，并输入下面的内容：

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

从命令行中，切换到 `hello.agda` 所在的文件夹内，然后运行：

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

你会看到如下的消息，而不是错误：

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

安装 PLFA 和 Agda 标准库

你也可以从 GitHub 克隆源码库，或者下载 Zip 包来获取最新版的《编程语言基础：Agda 语言描述》：

```
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 包括了所需要的 Agda 标准库版本，如果你在克隆时使用了 `--recurse-submodule` 选项，你在 `standard-library` 文件夹中已经有了 Agda 标准库！

如果你忘记使用了 `--recurse-submodules` 选项，也没有关系，我们可以修复它！

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

如果你用 Zip 包下载了 PLFA，你可以从 GitHub 上下载所需要的 Agda 标准库版本。 你可以从正确的分支克隆代码仓库，或者下载 Zip 包。

```
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.
```

最后，我们需要让 Agda 知道如何找到标准库。 你需要知道标准库安装的目录。检查 `standard-library.agda-lib` 文件是否存在，并记录这个文件的路径。 你需要在 `AGDA_DIR` 创建两个配置文件。在 UNIX 和 macOS 平台，`AGDA_DIR` 默认为 `~/.agda`。在 Windows 平台，`AGDA_DIR` 一般默认为 `%AppData%\agda`，而 `%AppData%` 默认为 `C:\Users\USERNAME\AppData\Roaming`。

- 如果 `AGDA_DIR` 文件夹不存在，创建它。
- 在 `AGDA_DIR` 中,创建一个纯文本文件 `libraries`,内容为 `/path/to/standard-library.agda-lib`（即上文中记录的路径）。 这个文件让 Agda 知道有一个名为 `standard-library` 的库可用。
- 在 `AGDA_DIR` 中，创建一个纯文本文件 `defaults`，内容仅为 `standard-library` 这一行。

关于放置标准库的更多信息可以参阅 Agda 文档中的库管理。

PLFA 也可以设置为一个 Agda 库。 如果你想完成 `courses` 目录中的习题，或者想导入书中的模块， 那么需要将 PLFA 设置为 Agda 库。完成此设置需要将 `plfa.agda-lib` 所在的路径作为单独的一行添加到 `AGDA_DIR/libraries`，并将 `plfa` 作为单独的一行添加到 `AGDA_DIR/defaults`。

检查 Agda 标准库是否正确安装

如果你愿意，你可以测试 Agda 标准库是否已正确。创建一个名为 `nats.agda` 的文件，包括以下内容：

```
open import Data.Nat

ten : ℕ
ten = 10
```

（注意 ℕ 是一个 Unicode 字符，而不是普通的大写字母 N。你可以直接从本页复制粘贴它到文件中。）

从命令行中，切换到 `nats.agda` 所在的文件夹内，然后运行：

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

你会看见数行来描述 Agda 在检查你的文件时载入的文件，但是没有错误：

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

设置 Agda 的编辑器

Emacs

推荐的 Agda 编辑器是 Emacs。安装 Emacs 可以用下面的方法：

- **UNIX 平台**：包管理器中的 Emacs 应该可以使用（只要它的版本比较新），GNU Emacs 下载页面也有最近发布版本的链接。
- **macOS 平台**：推荐的 Emacs 是 Aquamacs，但是 GNU Emacs 也可以通过 Homebrew 或者 MacPorts 安装。参阅 GNU Emacs 下载页面中的指示。
- **Windows 平台**：参阅 GNU Emacs 下载页面中的指示。

确保你可以用你安装的版本打开、编辑、保存文件。GNU Emacs 网站上的 Emacs 向导描述了如果打开 Emacs 安装中的教程。

Agda 自带了 Emacs 编辑器支持，如果你安装了 Agda，运行下面的命令来配置 Emacs：

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

如果你已经是 Emacs 用户，并有自己的设置，你会注意到 `setup` 命令向你的 `.emacs` 文件中追加了配置，来配合你已有的设置。

加上 `agda-mode`。Agda 自带了 `agda-mode`，因此如果你已经安装了 Agda，那么之需要运行以下命令就能配置好 `agda-mode` 了：

检查 `agda-mode` 是否正确安装

打开之前创建的 `nats.agda` 文件，使用 `C-c C-l` 来载入和类型检查这个文件。

在 Emacs 中自动加载 `agda-mode`

从版本 2.6.0 开始，Agda 支持 Markdown 风格的文学编程，文件使用 `.lagda.md` 扩展名。该扩展名的一个副作用就是大部分编辑器默认会进入 Markdown 编辑模式。而为了让 `agda-mode` 在你打开 `.agda` 或 `.lagda.md` 文件时自动加载， 请将以下内容放到你的 Emacs 配置文件中：

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

如果你配置中已有了改变 `auto-mode-alist` 的设置，将上述内容放置在已有的设置**之后**，或者将其与已有设置合并（如果你对 Emacs Lisp 足够了解）。 Emacs 的配置文件通常位于`~/.emacs` 或`~/.emacs.d/init.el`，然而 Aquamacs 用户需要将启动设置放到位于`~/Library/Preferences/Aquamacs Emacs/Preferences` 的 `Preferences.el` 文件中。 对于 Windows 平台，请参阅 GNU Emacs 文档 来寻找配置文件的位置。

可选：在 Emacs 中使用 Mononoki 字体

Agda 中的很多重要符号是用 Unicode 来表示的，因此用来显示和编辑 Agda 的字体需要正确地显示这些符号。最重要的是你使用的字体需要有好的 Unicode 字符支持。我们推荐 Mononoki， 其他的备选字体有 Source Code Pro、DejaVu Sans Mono 和 FreeMono。

你可以直接从 GitHub 下载并安装 Mononoki。对于大多数系统来说，安装字体只是简单的下载`.otf` 或者`.ttf` 文件。 如果你的包管理器提供了 Mononoki 的包，那样可能更加简单。 例如，macOS 的 Homebrew 在`cask-fonts` cask 中提供了 `font-mononoki` 包；Debian 的 APT 提供了`fonts-mononoki` 包。 将下面的内容加入 Emacs 配置文件，可以把 Mononoki 设置为 Emacs 的默认字体：

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

在 Emacs 中使用 agda-mode

要加载文件并对其执行类型检查，请使用`C-c C-l`。

Agda 的编辑是通过使用「洞」来交互的，它表示程序中尚未填充的片段。如果用问号作为表达式，并用`C-c C-l` 加载缓冲区，Agda 会将问号替换为一个「洞」。 当光标在洞中时，你可以做以下这些事情：

- `C-c C-c`：分项（询问变量名，**c**ase）
- `C-c C-空格`： 填洞
- `C-c C-r`： 用构造子精化（**r**efine）
- `C-c C-a`： 自动填洞（**a**utomatic）
- `C-c C-,`： 目标类型和上下文
- `C-c C-.`： 目标类型，上下文，以及推断的类型

更多细节请见 emacs-mode 文档。

如果你想在 Agda 代码的侧边而非底栏查看信息，那么可以这样做：

- 打开 Agda 文件并按 `C-c C-l`；
- 按 `C-x 1` 来仅显示当前 Agda 文件；
- 按 `C-x 3` 来垂直分割窗口；

- 将光标移动到右侧窗口；
- 按 `C-x b` 并输入「Agda information」切换到该缓冲区。

现在，Agda 的错误消息将会在代码右侧显示，而非被挤压在底部的狭小空间里。

在 Emacs 中使用 `agda-mode` 输入 Unicode 字符

当你书写 Agda 代码时，你需要输入标准键盘上没有的字符。Emacs 的「Agda-mode」定义了字符翻译来让这件事更容易：当你输入特定普通字符的序列时，Emacs 会在 Agda 文件中使用对应的特殊字符来替换。

例如，我们可以在之前的 `.agda` 测试文件中加入一条注释。比如说，我们想要加入下面的注释：

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

前几个字符都是普通字符，我们可以如往常的方式输入它们……

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

在最后一个空格之后，我们无法在键盘上找到 ∀ 这个键。这个字符的输入序列是四个字符 `\all`，所以我们输入这四个字符，当我们完成时，Emacs 会把它们替换成我们想要的……

```
{- I am excited to type ∀
```

我们继续输入其他字符。有时字符会在输入时发生变化，因为一个字符的输入序列是另一个字符输入序列的前缀。在我们输入箭头时会出现这样的情况，它的输入序列是 `\->`。在输入 `\-` 之后我们会看到……

```
{- I am excited to type ∀ and
```

……因为输入序列 `\-` 对应了一定长度的短横。当我们输入 `>` 时，‗ 变成了 `→`！ `≤` 的输入序列是 `\<=`，`≡` 的是 `\==`。

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

最后几个字符又回归了普通字符……

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

如果你在 Emacs 中输入 Unicode 时遇到了问题，可以在每章的最后找到当前章节中用到的 Unicode 字符。

带有 `agda-mode` 的 Emacs 提供了很多有用的命令，其中有两个对处理 Unicode 字符时特别有用。 支持的字符的完整列表可使用 `agda-input-show-translations` 命令查看：

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

`agda-mode` 支持的所有字符都会列出来。

如果你想知道如何在 agda 文件输入一个特定的 Unicode 字符，请将光标移动到该字符上并输入以下命令：

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

你会在迷你缓冲区中看到该字符对应的按键序列。

Spacemacs

Spacemacs 是一个「社区引领的 Emacs 版本」，对 Emacs 和 Vim 的编辑方式都有很好的支持。它自带集成了 `agda-mode`，所需的只是将它打开。

Visual Studio Code

Visual Studio Code 是一个微软开发的开源代码编辑器。 Visual Studio 市场中有 Agda 插件。

Atom

Atom 是一个 GitHub 开发的开源代码编辑器。 Atom 包管理器中有 Agda 插件。

开发者依赖

PLFA 是以 Pandoc Markdown 格式的文学化 Agda 编写的。 PLFA 可同时在网站上和作为 EPUB 电子书来阅读，二者均可在 UNIX 和 macOS 上构建。最后，为了帮助开发者避免常见错误，我们提供一系列 Git 钩子脚本。

构建网站和电子书

如果你想在本地构建 PLFA 的网站版，你只需要 Stack！ PLFA 用 Hakyll （一个构造静态网页的 Haskell 库）来构建。 我们配置了一个用于执行常见任务的 Makefile。例如，要构建 PLFA，运行：

```
make build
```

如果你想在本地架起 PLFA，并且在源文件修改时重新构造，润性：

```
make watch
```

Makefile 除了构造和监视之外，还提供了下列有用的其他功能：

```
build                       # 构造 PLFA
watch                       # 构造并架起 PLFA，监视文件变更并重新构造
test                        # 测试网页版是否有坏链、不合法的 HTML 等
test-epub                   # 测试 EPUB 电子书版是否符合 EPUB3 标准
clean                       # 清理 PLFA 构造
init                        # 设置 Git 钩子脚本（见下文）
update-contributors         # 从 GitHub 中拉取新贡献者信息至 contributors/
list                        # 列出所有构造功能
```

完整来说，Makefile 还提供了下列选项，但你可能不会需要用到：

```
legacy-versions             # 构造老版本的 PLFA
setup-install-bundler       # 安装 Ruby Bundler （‘legacy-versions’ 所需要）
setup-install-htmlproofer   # 安装 HTMLProofer （‘test’ 和 Git 钩子脚本所需要）
setup-check-fix-whitespace # 检查 fix-whitespace 是否已安装 （Git 钩子脚本所需要）
setup-check-epubcheck       # 检查 epubcheck 是否已安装 （EPUB 测试所需要）
setup-check-gem             # 检查 RubyGems 是否已安装
setup-check-npm             # 检查 Node 包管理器是否已安装
setup-check-stack           # 检查 Haskell 工具 Stack 是否已安装
```

本书的 EPUB 电子书版本会随着网站一起构造，因为网站上包括了电子书版本。

Git 钩子脚本

本源码库包含了几个 Git 钩子：

1. fix-whitespace 程序用来检查错误的空格。
2. 可运行的测试套件用来保证一切都能通过类型检查。

你可以运行 `make init` 来安装这些 Git 钩子。你可以运行以下命令来安装 fix-whitespace：

```
stack install fix-whitespace
```

如果你想让 Stack 使用你系统中安装的 GHC，那么可以传入 `--system-ghc` 参数并选择对应的 `stack-*.yaml` 文件，就像在安装 Agda 时那样。

Part I

第一分册：逻辑基础

Chapter 1

Naturals: 自然数

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

夜空中的星星不计其数，但只有不到五千颗是肉眼可见的。可观测宇宙中则包含大约 7*10^22 颗恒星。

星星虽多，但却是有限的，而自然数是无限的。就算用自然数把所有的星星都数尽了，剩下的自然数也和开始的一样多。

自然数是一种归纳数据类型（Inductive Datatype）

大家都熟悉自然数，例如：

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

等等。我们将自然数的**类型**（**Type**）记作 `ℕ` ，并称 `0` 、 `1` 、 `2` 、 `3` 等数 是类型 `ℕ` 的**值**（**Value**），表示为 `0 : ℕ` ， `1 : ℕ` ， `2 : ℕ` ， `3 : ℕ` 等等。

自然数集是无限的，然而其定义只需寥寥几行即可写出。下面是用两条**推导规则**（**Inference Rules**）定义的自然数：

```
--------
zero : ℕ

m : ℕ
---------
suc m : ℕ
```

以及用 Agda 定义的自然数：

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

其中 `ℕ` 是我们定义的**数据类型（Datatype）**的名字，而 `zero`（零）和 `suc`（**后继**，即 **Successor** 的简写）是该数据类型的**构造子（Constructor）**。

这两种定义说的是同一件事：

- **起始步骤（Base Case）**：`zero` 是一个自然数。
- **归纳步骤（Inductive Case）**：如果 `m` 是一个自然数，那么 `suc m` 也是。

此外，这两条规则给出了产生自然数**唯**一的方法。因此，可能的自然数包括：

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

我们将 `zero` 简写为 `0`；将 `suc zero`，零的后继数，也就是排在零之后的自然数，简写为 `1`；将 `suc (suc zero)`，也就是 `suc 1`，即一的后继数，简写为 `2`；将二的后继数简写为 `3`；以此类推。

练习 `seven`（实践）

请写出 `7` 的完整定义。

```
-- 请将代码写在此处。
```

推导规则分析

我们来分析一下刚才的两条推导规则。每条推导规则包含写在一条水平直线上的零条或多条**判断（Judgment）**，称之为**假设（Hypothesis）**；以及写在直线下的一条判断，称之为**结论（Conclusion）**。第一条规则是起始步骤：它没有任何假设，其结论断言 `zero` 是一个自然数。第二条规则是归纳步骤：它有一条假设，即 `m` 是自然数，而结论断言 `suc m` 也是一个自然数。

Agda 定义分析

现在分析一下 Agda 的定义。关键字 `data` 表示这是一个归纳定义，也就是用构造子定义一个新的数据类型。

```
ℕ : Set
```

表示 `ℕ` 是新的数据类型的名字，它是一个 `Set`，也就是在 Agda 中对类型的称呼。 关键字 `where` 用于分隔数据类型的声明和构造子的声明。每个构造子的声明独占一行，用缩进来指明它所属的 `data` 声明。

```
zero : ℕ
suc  : ℕ → ℕ
```

这两行给出了构造子 `zero` 和 `suc` 的类型**签名（Signature）**。 它们表示 `zero` 是一个自然数，`suc` 则取一个自然数作为参数，返回另一个自然数。

读者可能注意到 `ℕ` 和 `→` 在键盘上没有对应的按键。它们是 **Unicode（统一码）**中的符号。每一章的结尾都会有本章节引入的 Unicode 符号列表，以及在 Emacs 编辑器中输入它们的方法。

创世故事

我们再看一下自然数的定义规则：

- **起始步骤（Base Case）**：`zero` 是一个自然数。
- **归纳步骤（Inductive Case）**：如果 `m` 是一个自然数，那么 `suc m` 也是。

等等！第二行用自然数定义了自然数，这怎么能行？这个定义难道不会像「脱欧即是脱欧」一样无用吗？

【译注：「脱欧即是脱欧」是英国首相特蕾莎·梅提出的一句口号。】

实际上，不必通过自我指涉，我们的自然数定义也是可以被赋予意义的。我们甚至只需要处理**有限**的集合，而不必涉及**无限**的自然数集。

我们可以将这个过程比作一个创世故事。起初，我们对自然数一无所知：

```
-- 起初，世上没有自然数。
```

现在，我们对所有已知的自然数应用之前的规则。起始步骤告诉我们 `zero` 是一个自然数，所以我们将它加入已知自然数的集合中。归纳步骤告诉我们如果「昨天的」`m` 是一个自然数，那么「今天的」`suc m` 也是一个自然数。我们在今天之前并不知道任何自然数，所以归纳步骤在此处不适用。

```
-- 第一天，世上有了一个自然数。
zero : ℕ
```

然后我们重复此过程。今天我们知道昨天的所有自然数，以及任何通过规则添加的数。起始步骤依然告诉我们 `zero` 是一个自然数，我们已经知道这件事了。而如今归纳步骤告诉我们，由于 `zero` 在昨天是自然数， 那么 `suc zero` 在今天也是自然数：

```
-- 第二天，世上有了两个自然数。
zero : ℕ
suc zero : ℕ
```

我们再次重复此过程。现在归纳步骤告诉我们，由于 `zero` 和 `suc zero` 都是自然 数，因此 `suc zero` 和 `suc (suc zero)` 也是自然数。我们已经知道 `suc zero` 是自然数了，而后者 `suc (suc zero)` 是新加入的。

```
-- 第三天，世上有了三个自然数。
zero : ℕ
suc zero : ℕ
suc (suc zero) : ℕ
```

此时规律已经很明显了。

```
-- 第四天，世上有了四个自然数。
zero : ℕ
suc zero : ℕ
suc (suc zero) : ℕ
suc (suc (suc zero)) : ℕ
```

此过程可以继续下去。在第 *n* 天会有 *n* 个不同的自然数。每个自然数都会在 某一天出现。具体来说，自然数 *n* 在第 *n+1* 天首次出现。至此，我们并没有使用自然数集来定义其自身，而是根据第 *n* 天的数集定义了第 *n+1* 天的数集。

像这样的过程被称作是**归纳的 (Inductive)**。我们从一无所有开始，通过应用将一个有限集合转换到另一个有限集合的规则，逐步生成潜在无限的集合。

定义了零的规则之所以被称作**起始步骤**，是因为它在我们还不知道其它自然数时就引入了一个自然数。定义了后继数的规则之所以被称作**归纳步骤**，则是因为它在已知自然数的基础上引入了更多自然数。其中，起始步骤的重要性不可小觑。如果只有归纳步骤，那么第一天就没有任何自然数。第二天，第三天，无论多久也依旧没有。一个没有起始步骤的归纳定义是无用的，就像「脱欧即是脱欧」一样。

哲学和历史

哲学家发现，我们对「第一天」「第二天」等说法的使用暗含了对自然数的理解。在这个层面上，我们对自然数的定义也许某种程度上可以说是循环的，但我们不必为此烦恼。每个人对自然数都有良好的非形式化的理解，而我们可以将它作为形式化描述自然数的基础。

尽管从人类开始计数起，自然数就被人所认知，然而其归纳定义却是近代的事情。这可以追溯到理查德·戴德金（Richard Dedekind）于 1888 年发表的论文 *Was sind und was sollen die Zahlen?*“（《数是什么，应该是什么?》），以及朱塞佩·皮亚诺（Giuseppe Peano）于次年发表的著作”*Arithmetices principia, nova methodo exposita*”（《算术原理：用一种新方法呈现》）。

编译指令

在 Agda 中，任何跟在 `--` 之后或者由 `{-` 和 `-}` 包裹的文字都会被视为**注释（Comment）**。一般的注释对代码没有任何作用，但有一种例外，这种注释被称作**编译指令（Pragma）**，由 `{-#` 和 `#-}` 包裹。

```
{-# BUILTIN NATURAL ℕ #-}
```

这一行告诉 Agda 数据类型 `ℕ` 对应了自然数，然后编写者就可以将 `zero` 简写 为 `0`，将 `suc zero` 简写为 `1`，将 `suc (suc zero)` 简写为 `2` 了，以此类推。 必须要向编译指令给出之前声明的类型（本例中为`ℕ`），该类型有且只有两个构造子， 其中一个没有参数（本例中为`zero`），另一个只接受一个所给定类型的参数（本例中为`suc`）。

在启用上述简写的同时，这条编译指令也会用 Haskell 的任意精度整数类型来提供更高效的自然数内部表示。用 `zero` 和 `suc` 表示自然数 *n* 要占用正比 于 *n* 的空间，而将其表示为 Haskell 中的任意精度整数只会占用正比于 *n* 的对数的空间。

导入模块

我们很快就能写一些包含自然数的等式了。在此之前，我们需要从 Agda 标准库中导入**相等性（Equality）**的定义和用于等式推理的记法：

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

第一行代码将标准库中定义了相等性的模块导入到当前**作用域（Scope）**中，并将其命名为 `Eq`。第二行打开了这个模块，也就是将所有在 `using` 从句中指定的名称添加到当前作用域中。此处添加的名称有相等性运算符 `_≡_` 和两个项相等的证据 `refl`。第三行选取的模块提供了用于等价关系推理的运算符，并将 `using` 从句中指定的名称添加到当前作用域。此处添加的名称有 `begin_`、 `_≡⟨⟩_` 和 `_∎`。我们后面会看到这些运算符的使用方法。现在暂且把它们当作现成的工具来使用，不深究其细节。但我们会在相等性一章中学习它们的具体定义。

Agda 用下划线来标注**项（Term）**在中缀（Infix）或混缀（Mixfix）运算符中项出现的位置。因此，`_≡_` 和 `_≡⟨⟩_` 是中缀的（运算符写在两个项之间），而 `begin_` 是前缀的（运算符写在项之前），`_∎` 则是后缀的（运算符写在项之后）。

括号和分号是少有的几个不能在名称中出现的的字符，于是我们在 `using` 列表中不需要额外的空格来消除歧义。

自然数的运算是递归函数

既然我们有了自然数，那么可以用它们做什么呢？比如，我们能定义加法和乘法之类的算术运算吗？

我儿时曾花费了大量的时间来记忆加法表和乘法表。最开始，运算规则看起来很复杂，我也经常犯错。在发现**递归（Recursion）**时，我如同醍醐灌顶。有了这种简单的技巧，无数的加法和乘法运算只用几行就能概括。

这是用 Agda 编写的加法定义：

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

我们来分析一下它的定义。加法是一种中缀运算符，其名为 `_+_`，其中参数的位置用下划线表示。第一行指定了运算符的类型签名。类型 `ℕ → ℕ → ℕ` 表示加法接受两个自然数作为参数，并返回一个自然数。中缀记法只是函数应用的简写，`m + n` 和 `_+_ m n` 这两个项是等价的。

它的定义包含一个起始步骤和一个归纳步骤，与自然数的定义对应。起始步骤说明零加上一个数仍返回这个数，即 `zero + n` 等于 `n`。归纳步骤说明一个数的后继数加上另一个数返回两数之和的后继数，即 `(suc m) + n` 等于 `suc (m + n)`。在加法的定义中，构造子出现在了等式左边，我们称之为**模式匹配（Pattern Matching）**。

如果我们将 `zero` 写作 `0`，将 `suc m` 写作 `1 + m`，上面的定义就变成了 两个熟悉的等式。

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

因为零是加法的幺元，所以第一个等式成立。又因为加法满足结合律，所以第二个等式也成立。加法结合律的一般形式如下，说明运算结果与括号位置无关。

```
(m + n) + p  ≡  m + (n + p)
```

将上面第三个等式中的 `m` 换成 `1`，`n` 换成 `m`，`p` 换成 `n`，我们就得到了第二个等式。我们用等号 `=` 表示定义，用 `≡` 断言两个已定义的事物相等。

加法的定义是**递归（Recursive）**的，因为在最后一行我们用加法定义了加法。与自然数的归纳定义类似，这种表面上的循环性并不会造成问题，因为较大的数相加是用较小的数相加定义的。这样的定义被称作是**良基的（Well founded）**。

例如，我们来计算二加三：

```
_ : 2 + 3 ≡ 5
_ =
  begin
    2 + 3
  ≡⟨⟩ -- 展开为
    (suc (suc zero)) + (suc (suc (suc zero)))
  ≡⟨⟩ -- 归纳步骤
    suc ((suc zero) + (suc (suc (suc zero))))
  ≡⟨⟩ -- 归纳步骤
```

```
    suc (suc (zero + (suc (suc (suc zero)))))
  ≡⟨⟩ -- 起始步骤
    suc (suc (suc (suc (suc zero))))
  ≡⟨⟩ -- 简写为
    5
  ∎
```

我们可以按需展开简写，把同样的推导过程写得更加紧凑。

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

第一行取 `m = 1` 和 `n = 3` 匹配了归纳步骤，第二行取 `m = 0` 和 `n = 3` 匹配了归纳步骤，第三行取 `n = 3` 匹配了起始步骤。

以上两个推导过程都由一个类型签名（包含冒号 `:` 的一行）和一个提供对应类型 的项的绑定（Binding）（包含等号 `=` 的一行及之后的部分）组成。在编写代码时 我们用了虚设名称 `_`。虚设名称可以被重复使用，在举例时非常方便。除了 `_` 之外 的名称在一个模块里只能被定义一次。

这里的类型是 `2 + 3 ≡ 5`，而该等式写成列表形式的等式链的项，提供了类型中表示等式成立的**证据（Evidence）**。该等式链由 `begin` 开始，以 `∎` 结束（`∎` 可读作「qed（证毕）」或「tombstone（墓碑符号）」，后者来自于其外观），并由一系列 `≡⟨⟩` 分隔的项组成。

其实，以上两种证明都比实际所需的要长，下面的证明就足以让 Agda 满意了。

```
_ : 2 + 3 ≡ 5
_ = refl
```

Agda 知道如何计算 `2 + 3` 的值，也可以立刻确定这个值和 `5` 是一样的。如果一个 二元关系（Binary Relation）中每个值都和自己相关，我们称这个二元关系满足**自反性（Reflexivity）**。在 Agda 中，一个值等于其自身的证据写作 `refl`。

在等式链中，Agda 只检查每个项是否都能化简为相同的值。如果我们打乱等式顺序，省略或者加入一些额外的步骤，证明仍然会被接受。我们需要自己来保证等式的顺序便于理解。

在这里，`2 + 3 ≡ 5` 是一个类型，等式链（以及`refl`）都是这个类型的项。 换言之，我们也可以把每个项都看作断言`2 + 3 ≡ 5`的**证据**。这种解释的对偶性——类型即命题，项即证据——是我们在 Agda 中形式化各种概念的核心， 也是贯穿本书的主题。

注意，当我们使用**证据**这个词时不容一点含糊。这里的证据确凿不移，不像法庭上的证词一样必须被反复权衡以决定证人是否可信。我们也会使用**证明**一词表达相同的意思，在本书中这两个词可以互换使用。

练习`+-example`（实践）

计算 `3 + 4`，将你的推导过程写成等式链，为 `+` 使用等式。

```
-- 请将代码写在此处。
```

乘法

一旦我们定义了加法，我们就可以将乘法定义为重复的加法。

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

计算 `m * n` 返回的结果是 `m` 个 `n` 之和。

重写定义再一次给出了两个熟悉的等式：

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

因为零乘任何数都是零，所以第一个等式成立。因为乘法对加法有分配律，所以第二个等式也成立。乘法对加法的分配律的一般形式如下：

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

将上面第三个等式中的 `m` 换成 `1`，`n` 换成`m`，`p` 换成 `n`，再根据 一是乘法的幺元，也就是`1 * n ≡ n`，我们就得到了第二个等式。

这个定义也是良基的，因为较大的数相乘是用较小的数相乘定义的。

例如，我们来计算二乘三：

```
_ =
  begin
```

```
  2 * 3
≡⟨⟩ -- 归纳步骤
  3 + (1 * 3)
≡⟨⟩ -- 归纳步骤
  3 + (3 + (0 * 3))
≡⟨⟩ -- 起始步骤
  3 + (3 + 0)
≡⟨⟩ -- 化简
  6
∎
```

第一行取 `m = 1` 和 `n = 3` 匹配了归纳步骤，第二行取 `m = 0` 和 `n = 3` 匹配了归纳步骤，最后第三行取 `n = 3` 匹配了起始步骤。在这里我们省略了 `_ : 2 * 3 ≡ 6` 的签名，因为它很容易从对应的项推导出来。

练习 `*-example`（实践）

计算 `3 * 4`，将你的推导过程写成等式链，为 `*` 使用等式。（不必写出 `+` 求值的每一步。）

```
-- 请将代码写在此处。
```

练习 `_^_`（推荐）

根据以下等式写出乘方的定义。

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

检查 `3 ^ 4` 是否等于 `81`。

```
-- 请将代码写在此处。
```

饱和减法

我们也可以定义减法。由于没有负的自然数，如果被减数比减数小，我们就将结果取零。这种针对自然数的减法变种称作**饱和减法（Monus，由 minus 修改而来）**。

饱和减法是我们首次在定义中对两个参数都使用模式匹配：

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

我们可以通过简单的分析来说明所有的情况都被考虑了。

- 考虑第二个参数。
  - 如果它是 `zero`，应用第一个等式。
  - 如果它是 `suc n`，考虑第一个参数。
    * 如果它是 `zero`，应用第二个等式。
    * 如果它是 `suc m`，应用第三个等式。

此也是良基的，因为较大的数的饱和减法是用较小的数的饱和减法定义的。

例如，我们来计算三减二：

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

我们没有使用第二个等式，但是如果被减数比减数小，我们还是会用到它。

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

练习 `∸-example₁` 和 `∸-example₂`（推荐）{name=monus-examples}

计算 `5 ∸ 3` 和 `3 ∸ 5`，将你的推导过程写成等式链。

```
-- 请将代码写在此处。
```

优先级

我们经常使用**优先级（Precedence）**来避免书写大量的括号。函数应用比其它任何运算符都**结合得更紧密**（即**有更高的优先级**），所以我们可以用 `suc m + n` 来表示 `(suc m) + n`。另一个例子是，我们说乘法比加法结合得更紧密，所以可以用 `n + m * n` 来表示 `n + (m * n)`。我们有时候也说加法是**左结合的**，所以可以用 `m + n + p` 来表示 `(m + n) + p`。

在 Agda 中，中缀运算符的优先级和结合性需要被声明：

```
infixl 6 _+_ _∸_
infixl 7 _*_
```

它声明了运算符 `_+_` 和 `_∸_` 的优先级为 6，运算符 `_*_` 的优先级 为 7。因为加法和饱和减法的优先级更低，所以它们结合得不如乘法紧密。`infixl` 意味着三个运算符都是左结合的。编写者也可以用 `infixr` 来表示 某个运算符是右结合的，或者用 `infix` 来表示总是需要括号来消除歧义。

柯里化

我们曾将接受两个参数的函数表示成「接受第一个参数，返回接受第二个参数的函数」的函数。这种技巧叫做**柯里化（Currying）**。

与 Haskell 和 ML 等函数式语言类似，Agda 在设计时就考虑了让柯里化更加易用。函数箭头是右结合的，而函数应用是左结合的。

比如

`ℕ → ℕ → ℕ` 表示 `ℕ → (ℕ → ℕ)`

而

`_+_ 2 3` 表示 `(_+_ 2) 3`。

`_+_ 2` 这个项表示一个「将参数加二」的函数，因此将它应用到三就得到了五。

柯里化是以哈斯凯尔·柯里（Haskell Curry）的名字命名的，编程语言 Haskell 也是。 柯里的工作可以追溯到 19 世纪 30 年代。当我第一次了解到柯里化时，有人告诉我柯里化的命名是个归因错误，因为在 20 年代同样的想法就已经被 Moses Schönfinkel 提出了。我也听说过这样一个笑话：「（柯里化）本来该命名成 Schönfinkel 化的，但是咖喱（Curry）更好吃」。直到之后我才了解到，这个归因错误的解释本身也是个归因错误。柯里化的概念早在戈特洛布·弗雷格（Gottlob Frege）发表于 1879 年的 **“Begriffsschrift”**（《**概念文字**》）中就出现了。

又一个创世故事

和归纳定义中用自然数定义了自然数一样，递归定义也用加法定义了加法。

同理，无需利用循环性，我们的加法定义也是可以被赋予意义的。为此，我们需要将加法的定义规约到用于判断相等性的等价的推导规则上来。

```
n ∶ ℕ
-------------
zero + n  =  n

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

假设我们已经定义了自然数的无限集合，指定了判断 `n ∶ ℕ` 的意义。第一条推导规则是起始步骤。它断言如果 `n` 是一个自然数，那么零加上它得 `n`。第二条推导规则是归纳步骤。它断言如果 `m` 加上 `n` 得 `p`，那么 `suc m` 加上 `n` 得 `suc p`。

我们同样借创世故事来帮助理解，不过这次关注的是关于加法的判断。

```
-- 起初，我们对加法一无所知。
```

现在对所有已知的判断应用之前的规则。起始步骤告诉我们，对于 每个自然数`n` 都有 `zero + n = n`，因此我们添加所有的这类等式。归纳步骤告诉我们，如果「昨天」有 `m + n = p`，那么「今天」就有 `suc m + n = suc p`。在今天之前，我们不知道任何关于加法的等式，因此这条规则不会给我们任何新的等式。

```
-- 第一天，我们知道了 0 为被加数的加法。
0 + 0 = 0    0 + 1 = 1    0 + 2 = 2    ...
```

然后我们重复这个过程。今天我们知道来自昨天的所有等式，以及任何通过规则添加的等式。起始步骤没有告诉我们任何新东西，但是归纳步骤添加了更多的等式。

```
-- 第二天，我们知道了 0, 1 为被加数的加法。
0 + 0 = 0    0 + 1 = 1    0 + 2 = 2    0 + 3 = 3    ...
1 + 0 = 1    1 + 1 = 2    1 + 2 = 3    1 + 3 = 4    ...
```

我们再次重复这个过程：

```
-- 第三天，我们知道了 0, 1, 2 为被加数的加法。
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    ...
```

此时规律已经很明显了：

```
-- 第四天，我们知道了 0, 1, 2, 3 为被加数的加法。
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    ...
```

此过程可以继续下去。在第 m 天我们将知道所有被加数小于 m 的等式。

如上所示，归纳定义和递归定义的的推导过程十分相似。它们就像一枚硬币的两面。

有限的创世故事

前面的创世故事是用分层的方式讲述的。首先，我们创造了自然数的无限集合。然后，我们构造加法的实例时把自然数集视为现成的，所以即使在第一天我们也有一个无限的实例集合。

然而，我们也可以选择同时构造自然数集和加法的实例。这样在任何一天都只会有一个有限的实例集合。

```
-- 起初，我们一无所知。
```

现在，对我们已知的所有判断应用之前的规则。只有自然数的起始步骤适用：

```
-- 第一天，我们知道了零。
0 ∶ ℕ
```

我们再次应用所有的规则。这次我们有了一个新自然数，和加法的第一个等式。

```
-- 第二天，我们知道了一和所有和为零的加法算式。
0 ∶ ℕ
1 ∶ ℕ    0 + 0 = 0
```

然后我们重复这个过程。我们通过加法的起始步骤得到了一个等式，也通过在前一天的等式上应用加法的归纳步骤得到了一个等式：

```
-- 第三天，我们知道了二和所有和为一的加法算式。
0 ∶ ℕ
1 ∶ ℕ    0 + 0 = 0
2 ∶ ℕ    0 + 1 = 1   1 + 0 = 1
```

此时规律已经很明显了：

```
-- 第四天，我们知道了三和所有和为二的加法算式。
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
```

在第 *n* 天会有 *n* 个不同的自然数和 *n* × *(n-1) / 2* 个加法等式。 数字 *n* 和所有和小于 *n* 的加法等式在第 *n+1* 天首次出现。这提供了一种无限的数据集合及与之相关的等式的有限主义视角。

交互式地编写定义

Agda 被设计为使用 Emacs 作为文本编辑器，二者一同提供了很多能帮助用户交互式地创建定义和证明的功能。

我们从输入以下代码开始：

```
_+_ ꞉ ℕ → ℕ → ℕ
m + n = ?
```

问号表示你希望 Agda 帮助你填入这部分代码。如果按下组合键 `C-c C-l` （按住 Control 键的同时先按 `c` 键再按 `l` 键，`l` 键代表载入 **l**oad），这个问号会被替换：

```
_+_ ꞉ ℕ → ℕ → ℕ
m + n = { }0
```

这对花括号被称作一个**洞（Hole）**，0 是这个洞的编号。洞将会被高亮显示为 绿色（或蓝色）。同时，Emacs 会创建一个窗口显示如下文字：

```
?0 ꞉ ℕ
```

这表示 0 号洞需要填入一个类型为 `ℕ` 的项。按组合键 `C-c C-f` （`f` 键代表向前 **f**orward）会移动到下一个洞。

我们希望在第一个参数上递归来定义加法。 将光标移至 0 号洞并按 `C-c C-c` （`c` 键代表分情况讨论 **c**ase），你将看见如下提示：

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

即「用于分项的模式变量（留空以对结果分项）：」。

键入 `m` 会对名为 `m` 的变量分项（即自动模式匹配），并将代码更新为：

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

现在有两个洞了。底部的窗口会告诉你每个洞所需的类型：

```
?0 : ℕ
?1 : ℕ
```

移动至 0 号洞，按下 `C-c C-,` 会显示当前洞所需类型的具体信息，以及 可用的自由变量：

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

这些信息强烈建议了用 `n` 填入该洞。填入内容后，你可以按下`C-c C-空格` 来移除这个洞。

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

同理，移动到 1 号洞并按下 `C-c C-,` 会显示当前洞所需类型的具体信息， 以及可用的自由变量：

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

移动到一个洞并按下 `C-c C-r` （`r` 键表示细分 **r**efine）会将一个构造子填入这个洞（如果有唯一的选择的话），或者告诉你有哪些可用的构造子以供选择。在当前情况下，编辑器会显示如下内容：

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

即“不知道在 `zero` 和 `suc` 中该引入哪一个构造子”。

我们将 `suc ?` 填入并按下`C-c C-空格`，它会将代码更新为：

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

移动到新的洞并按下 `C-c C-,` 给出了和之前类似的信息：

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

我们可以用 `m + n` 填入该洞并按 `C-c C-空格` 来完成程序：

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

在如此简单的程序上频繁使用交互操作可能帮助不大，但是同样的技巧能够帮助你构建更复杂的程序。甚至对于加法定义这么简单的程序，使用 `C-c C-c` 来分项仍然是有用的。

更多编译指令

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

以上几行告诉 Agda 这几个运算符和数学中常用的运算符相对应，以便让它在计算时使用相应的，可处理任意精度整数类型的 Haskell 运算符。计算 `m` 加 `n` 时，用 `zero` 和 `suc` 表示的自然数需要正比于 `m` 的时间，而用 Haskell 整数表示的情况下只需要正比于 `m` 和 `n` 中较大者的对数的时间。 类似地，计算 `m` 乘 `n` 时，用 `zero` 和 `suc` 表示的自然数需要正比于 `m` 乘 `n` 的 时间，而用 Haskell 整数表示的情况下只需要正比于 `m` 和 `n` 的对数之和的时间。

练习 `Bin`（拓展）

使用二进制系统能提供比一进制系统更高效的自然数表示。我们可以用一个比特串来表示一个数：

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

例如，以下比特串

```
1011
```

代表数字十一被编码为了

```
⟨⟩ I O I I
```

由于前导零的存在，表示并不是唯一的。因此，十一同样可以 表示成 `001011`，编码为：

```
⟨⟩ O O I O I I
```

定义这样一个函数

```
inc : Bin → Bin
```

将一个比特串转换成下一个数的比特串。比如，`1100` 编码了十二，我们就应该有：

```
inc (⟨⟩ I O I I) ≡ ⟨⟩ I I O O
```

实现这个函数，并验证它对于表示零到四的比特串都能给出正确结果。

使用以上的定义，再定义一对函数用于在两种表示间转换。

```
to   : ℕ → Bin
from : Bin → ℕ
```

对于前者，用没有前导零的比特串来表示正数，并用 `⟨⟩ O` 表示零。验证这两个函数都能对零到四给出正确结果。

```
-- 请将代码写在此处。
```

标准库

在每一章的结尾，我们将展示如何在标准库中找到相关的定义。自然数，它们的构造子，以及用于自然数的基本运算符，都在标准库模块 `Data.Nat` 中定义：

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

正常情况下，我们会以运行代码的形式展示一个导入语句，这样如果我们尝试导入一个不可用的定义，Agda 就会报错。不过现在，我们只在注释里展示了这个导入语句。这一章和标准库 都调用了 `NATURAL` 编译指令。我们是在 `ℕ` 上使用，而标准库是在等价的类型 `Data.Nat.ℕ` 上使用。这样的编译指令只能被调用一次，因为 重复调用会导致 `2` 到底是类型 `ℕ` 的值还是类型 `Data.Nat.ℕ` 的值这样的困惑。重复调用其它的编译指令也会导致同样的问题。基于这个原因，我们在后续章节中通常会避免使用编译指令。更多关于编译指令的信息可在 Agda 文档中找到。

Unicode

这一章使用了如下的 Unicode 符号:

```
ℕ  U+2115  双线体大写 N (\bN)
→  U+2192  右箭头 (\to, \r, \->)
∸  U+2238  点减 (\.-)
≡  U+2261  等价于 (\==)
⟨  U+27E8  数学左尖括号 (\<)
⟩  U+27E9  数学右尖括号 (\>)
∎  U+220E  证毕 (\qed)
```

以上的每一行均包含 Unicode 符号（如 `ℕ`），对应的 Unicode 码点（如 `U+2115`）， 符号的名称（如 `双线体大写 N`），以及用于在 Emacs 中键入该符号的按键序列（如 `\bN`）。

通过 `\r` 命令可以查看多种右箭头符号。在输入 `\r` 后，你可以按左、右、上、下键来查看或选择可用的箭头符号。这个命令会记住你上一次选择的位置，并在下一次使用时从该字符开始。用于输入左箭头的 `\l` 命令的用法与此类似。

除了在输入箭头的命令中使用左、右、上、下键以外，以下按键也可以起到相同的作用:

```
C-b  左 (后退一个字符)
C-f  右 (前进一个字符)
C-p  上 (到上一行)
C-n  下 (到下一行)
```

`C-b` 表示按 Control + b，其余同理。你也可以直接输入显示的列表中的数字编号来选择。

要查看所支持字符的完整列表，请执行 `agda-input-show-translations` 命令:

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

这样会显示出 `agda-mode` 中所有支持的字符。我们用 M-x 表示按下 `ESC` 后再按下 `x`。

如果你想知道如何在 agda 文件中输入一个特定的 Unicode 字符，请将光标移至该字符上， 然后执行 `quail-show-key` 命令:

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

你会在迷你缓冲区中看到输入该字符所需的按键序列。 例如，如果你在 `∸` 上执行 `M-x quail-show-key`，就会看到该字符的按键序列为 `\.-`。

Chapter 2

Induction: 归纳证明

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

> 归纳会让你对无中生有感到内疚……但它却是文明中最伟大的思想之一。 —— Herbert Wilf

现在我们定义了自然数及其运算，下一步是学习如何证明它们满足的性质。顾名思义，**归纳数据类型（Inductive Datatype）**是通过**归纳（Induction）**来证明的。

导入

我们需要上一章中的相等性，加上自然数及其运算。我们还导入了一些新的运算：`cong`、`sym` 和 `_≡⟨_⟩_`，之后会解释它们：

```
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; _+_; _*_; _∸_)
```

运算符的性质

运算符随处可见，而数学家们统一了一些最常见的性质的名称。

- **幺元（Identity）**：对于所有的 `n`，若 `0 + n ≡ n`，则 `+` 有左幺元 `0`；若 `n + 0 ≡ n`，则 `+` 有右幺元 `0`。同时为左幺元和右幺元的值称简称幺元。幺元有时也称作**单位元（Unit）**。
- **结合律（Associativity）**：若括号的位置无关紧要，则称运算符 `+` 满足结合律，即对于所有的 `m`、`n` 和 `p`，有 `(m + n) + p ≡ m + (n + p)`。

- **交换律（Commutativity）**：若参数的顺序无关紧要，则称运算符 `+` 满足交换律，即对于所有的 `m` 和 `n`，有 `m + n ≡ n + m`。
- **分配律(Distributivity)**:对于所有的 `m`、`n` 和 `p`,若 `(m + n) * p ≡ (m * p) + (n * p)`,则运算符 `*` 对运算符 `+` 满足左分配律；对于所有的 `m`、`n` 和 `p`,若 `m * (p + q) ≡ (m * p) + (m * q)`,则满足右分配律。

加法的幺元为 `0`，乘法的幺元为 `1`。加法和乘法都满足结合律和交换律， 乘法对加法满足分配律。

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.

如果你在一个舞会上碰见了一位操作员，那么你可以跟他闲聊，问问他是否有单位元，能不能结合或者交换。如果你碰见了两位操作员，那么可以问他们某一位是否在另一位上面分布。

【译注：作者的双关冷笑话，运算符（Operator）也有操作员的意思。】

正经来说，如果你在阅读技术论文时遇到了一个运算符，那么你可以考察它是否拥有幺元，是否满足结合律或分配律，或者是否对另一个运算符满足分配律，这能为你提供一种视角。细心的作者通常会指出它们是否满足这些性质，比如说指明一个新引入的运算符满足结合律但不满足交换律。

练习 `operators`（实践）

请给出另一对运算符，它们拥有一个幺元，满足结合律、交换律，且其中一个对另一个满足分配律。（你不必证明这些性质）

```
-- 请将代码写在此处。
```

请给出一个运算符的例子，它拥有幺元、满足结合律但不满足交换律。（你不必证明这些性质）

```
-- 请将代码写在此处。
```

结合律

加法的一个性质是满足**结合律**，即括号的位置无关紧要：

```
(m + n) + p ≡ m + (n + p)
```

这里的变量 `m`、`n` 和 `p` 的取值范围都是全体自然数。

我们可以为这三个变量选取特定的数值来验证此命题：

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

在这里，我们将计算过程写成了等式链，每行一个式子。这样的等式链通常非常易读，你可以从上到下，直到遇到最简形式（本例中为`12`），也可以从下到上，直到回到同样的式子。

该测试揭示了结合律可能没有它初看起来那么显然。为什么`7 + 5`与`3 + 9`相同？我们可能需要收集更多证据，选择其它的数值来验证此命题。但由于自然数是无限的，因此测试永远无法完成。那么我们还有其它可以确保结合律对于**所有**自然数都成立的方法吗？

当然有！我们可以用**归纳证明（Proof by Induction）**来确保某个性质对于所有的自然数都成立。

归纳证明

回想自然数的定义，它由一个**起始步骤**「`zero`是一个自然数」和一个**归纳步骤**「若`m`是一个自然数，则`suc m`也是一个自然数」构成。

归纳证明遵循此定义的结构。要通过归纳证明自然数的某个性质，我们需要两个步骤。其一是**起始步骤**，即需要证明此性质对`zero`成立。其二是**归纳步骤**，即假设此性质对一个任意自然数`m`成立（我们称之为**归纳假设（Induction Hypothesis）**），然后证明该性质对`suc m`必定成立。

若将`m`的某种性质（Property）写作`P m`，那么我们需要证明的就是以下两个推导规则：

```
------
P zero

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

先来分析一下这些规则。第一条规则是起始步骤，它需要我们证明性质`P`对`zero`成立。第二条规则是归纳步骤，它需要我们证明若归纳假设「`P`对`m`成立」，那么`P`也对`suc m`成立。

为什么可以这样做呢？它也可以用创世故事来讲解。起初，我们对性质一无所知：

```
-- 起初，世上没有已知的性质。
```

现在我们对所有已知的性质应用上述两条规则。起始步骤告诉我们`P zero`成立，所以我们将它加入已知的性质集合中。归纳步骤告诉我们若「昨天的」`P m`成立，那么「今天的」`P (suc m)`也成立。我们在今天之前并不知道任何性质，因此归纳步骤在这里不适用：

```
-- 第一天，我们知道了一个性质。
P zero
```

然后我们重复此过程。在接下来的一天我们知道今天之前的所有性质，以及任何通过此规则添加的性质。起始步骤告诉我们`P zero`成立，我们已经知道这件事了。而如今归纳步骤告诉我们，由于`P zero`在昨天成立，那么`P (suc zero)`今天也成立。

```
-- 第二天，我们知道了两个性质。
P zero
P (suc zero)
```

我们再重复此过程。现在归纳步骤告诉我们由于`P zero`和`P (suc zero)`都成立，因此`P (suc zero)`和`P (suc (suc zero))`也成立。我们已经知道第一个成立了，但第二个是新引入的：

```
-- 第三天，我们知道了三个性质。
P zero
P (suc zero)
P (suc (suc zero))
```

此时规律已经很明显了：

```
-- 第四天，我们知道了四个性质。
P zero
P (suc zero)
P (suc (suc zero))
P (suc (suc (suc zero)))
```

此过程可以继续下去。在第 *n* 天会有 *n* 个不同的性质成立。每个自然数的性质都会在某一天出现。具体来说，性质`P n`会在第 *n+1* 天首次出现。

第一个证明：结合律

要证明结合律，我们需要将`P m`看做以下性质：

```
(m + n) + p ≡ m + (n + p)
```

这里的 `n` 和 `p` 是任意自然数，因此若我们可以证明该等式对所有的 `m` 都成立，那么它也会对所有的 `n` 和 `p` 成立。其推理规则的对应实例如下：

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

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

如果我们可以证明这两条规则，那么加法结合律就可以用归纳法来证明。

以下为此性质的陈述和证明：

```
+-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)
  ∎
```

我们将此证明命名为 `+-assoc`。在 Agda 中，标识符可以由除空格和 `@.(){};_` 之外的任何字符序列构成。

我们来分析一下这段代码。其签名（Signature）描述了我们定义的标识符 `+-assoc` 为以下命题提供了证据（Evidence）：

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

倒 A 符号读作「对于所有（for all）」，而该命题断言对于所有的自然数 `m`、`n` 和 `p`，等式 `(m + n) + p ≡ m + (n + p)` 成立。该命题的证据是一个接受三个自然数的函数，将它们绑定到 `m`、`n` 和 `p`，并返回该等式对应实例的证据。

对于起始步骤，我们必须证明：

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

用加法的起始步骤化简等式两边会得到：

```
n + p ≡ n + p
```

此式平凡成立。阅读此证明中起始步骤中的等式链，其最初和最末的式子分别匹配待证等式的两边，从上到下或从下到上读都会让我们在中间遇到 `n + p` 。此步骤无需多言，化简即可。

对于归纳步骤，我们必须证明：

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

用加法的归纳步骤化简等式两边会得到：

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

反之，它也可以通过在归纳假设

```
(m + n) + p ≡ m + (n + p)
```

两边之前加上 `suc` 得到。

阅读此证明中归纳步骤的等式链，其最初和最末的式子分别匹配待证等式的两边，从上到下或从下到上读都会让我们到达上面化简等式的地方。剩下的等式单化简还不行，我们还需要为推理链使用一个附加的运算符 `_≡⟨_⟩_`，并将等式的依据放在尖括号中。这里给出的依据是：

```
⟨ cong suc (+-assoc m n p) ⟩
```

在这里，递归调用的 `+-assoc m n p` 拥有归纳假设的类型，而 `cong suc` 会在等式两边的前面加上 `suc` 以得到需要的等式。

若某个关系在应用了给定的函数后仍然保持不变，则称该关系满足**合同性**（**Congruence**）。若 `e` 是 `x ≡ y` 的证据，那么对于任意函数 `f`， `cong f e` 就是 `f x ≡ f y` 的证据。

在这里并未假定归纳假设，而是通过递归调用我们定义的函数 `+-assoc m n p` 来证明。对于加法，这是良基的（well-founded），因为更大数值的结合律可基于更小数值的结合律来证明。在此步骤中，`assoc (suc m) n p` 是用 `assoc m n p` 证明的。 归纳证明和递归定义之间的这种对应是 Agda 中最吸引人的方面之一。

归纳即递归

下面是归纳如何对应于递归的具体例子，它是在结合律的证明中，将 `m` 实例化为 `2` 时的计算过程。

```
+-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)
      ∎
```

术语与记法

在结合律的陈述中出现的符号 `∀` 表示它对于所有的`m`、`n` 和 `p` 都成立。 我们将 `∀` 称为**全称量词**（Universal Quantifier），我们会在 Quantifiers 章节中进一步讨论。

全称量词的证据是一个函数。记法

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

和

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

是等价的。和 `ℕ → ℕ → ℕ` 这样的函数类型不同，上述函数中的变量与每一个实参类型相关联，且其结果类型可能会涉及（或依赖于）这些变量，因此它们叫做**依赖函数**（Dependent Function）。

第二个证明：交换律

加法的另一个重要性质是满足**交换律（Commutativity）**，即运算数的顺序无关紧要：

```
m + n ≡ n + m
```

要证明它，我们需要先证明两条引理（Lemma）。

第一条引理

加法定义的起始步骤说明零是一个左幺元：

```
zero + n ≡ n
```

我们的第一条引理则说明零也是一个右幺元：

```
m + zero ≡ m
```

以下是此引理的证明：

```
+-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
  ∎
```

其签名说明我们定义的标识符 `+-identityʳ` 提供了以下命题的证据：

```
∀ (m ⦂ ℕ) → m + zero ≡ m
```

该命题的证据是一个函数，它接受一个自然数，将其绑定到 `m`，然后返回 该等式对应实例的证据。它通过对 `m` 进行归纳来证明。

对于起始步骤，我们必须证明：

```
zero + zero ≡ zero
```

根据加法的起始步骤化简，这很显然。

对于归纳步骤，我们必须证明：

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

根据加法的归纳步骤化简等式两边可得：

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

反之，它也可以通过在归纳假设

```
m + zero ≡ m
```

两边之前加上 `suc` 得到。

阅读此等式链，从上到下和从下到上读都会让我们到达上面化简等式的地方。剩下的等式可由以下依据得出：

```
⟨ cong suc (+-identityʳ m) ⟩
```

在这里，递归调用的 `+-identityʳ m` 拥有归纳假设的类型，而 `cong suc` 会在等式两边的前面加上 `suc` 以得到需要的等式。第一条引理证毕。

第二条引理

加法定义的归纳步骤将第一个参数的 `suc` 推到了外面：

```
suc m + n ≡ suc (m + n)
```

我们的第二条引理则对第二个参数的 `suc` 做同样的事情：

```
m + suc n ≡ suc (m + n)
```

下面是该引理的陈述和证明：

```
+-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)
  ∎
```

其签名说明我们定义的标识符 `+-suc` 提供了以下命题的证据：

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

该命题的证据是一个函数，它接受两个自然数，将二者分别绑定到 `m` 和 `n`， 并返回该等式对应实例的证据。它通过对 `m` 进行归纳来证明。

对于起始步骤，我们必须证明：

```
zero + suc n ≡ suc (zero + n)
```

根据加法的起始步骤化简，这很显然。

对于归纳步骤，我们必须证明：

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

根据加法的归纳步骤化简等式两边可得：

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

反之，它也可以通过在归纳假设

```
m + suc n ≡ suc (m + n)
```

两边之前加上 `suc` 得到。

从上到下或从下到上阅读等式链都会让我们在中间遇到化简后的等式。剩下的等式可由以下依据得出：

```
⟨ cong suc (+-suc m n) ⟩
```

在这里，递归调用的 `+-suc m n` 拥有归纳假设的类型，而 `cong suc` 会在等式两边的前面加上 `suc` 以得到需要的等式。第二条引理证毕。

命题

最后，以下是我们的命题的陈述和证明：

```
+-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
  ∎
```

第一行说明我们定义的标识符 `+-comm` 提供了以下命题的证据：

```
∀ (m n ꞉ ℕ) → m + n ≡ n + m
```

该命题的证据是一个函数，它接受两个自然数，将二者分别绑定到 `m` 和 `n`， 并返回该等式对应实例的证据。它通过对 `n` 进行归纳来证明。（这次不是 `m`！）

对于起始步骤，我们必须证明：

```
m + zero ≡ zero + m
```

根据加法的起始步骤化简等式两边可得：

```
m + zero ≡ m
```

剩下的等式可由依据 `⟨ +-identityʳ m ⟩` 得出，它调用第一条引理。

对于归纳步骤，我们必须证明：

```
m + suc n ≡ suc n + m
```

根据加法的归纳步骤化简等式两边可得：

```
m + suc n ≡ suc (n + m)
```

我们分两步来证明它。首先，我们有：

```
m + suc n ≡ suc (m + n)
```

它依据第二条引理 `⟨ +-suc m n ⟩` 得出。之后我们有：

```
suc (m + n) ≡ suc (n + m)
```

它依据合同性和归纳假设 `⟨ cong suc (+-comm m n) ⟩` 得出。证毕。

Agda 要求标识符必须在使用前定义，因此我们必须在主命题之前列出引理，如前例所示。在实践中，我们通常会先试着证明主命题，之后所需的等式会说明需要证明哪些引理。

第一个推论：重排定理

我们可以随意应用结合律来重排括号。例如：

```
+-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
  ∎
```

无需归纳法，我们只不过应用了两次结合律就完成了证明。其中有几点需要注意的地方。

第一，加法是左结合的，因此 `m + (n + p) + q` 表示 `(m + (n + p)) + q`。

第二，我们用 `sym` 来交换等式的两边。命题 `+-assoc n p q` 会将括号从右边移到左边：

```
(n + p) + q ≡ n + (p + q)
```

要往另一个方向移动括号，我们要用 `sym (+-assoc m n p)`：

```
n + (p + q) ≡ (n + p) + q
```

一般来说，若 `e` 提供了 `x ≡ y` 的证据，那么 `sym e` 就提供了 `y ≡ x` 的证据。

第三，Agda 支持 Richard Bird 引入的**片段（Section）**记法。我们将应用到 `y` 并返回 `x + y` 的函数写作 `(x +_)`。因此，应用合同性 `cong (m +_)` 会将上面的等式转换成：

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

类似地，我们将应用到 `y` 并返回 `y + x` 的函数写作 `(_+ x)`。这同样适用于任何中缀运算符。

创世，最后一次

我们回到结合律的证明上来，把归纳证明（或等价的递归定义）看做一个创世故事会有助于理解。这次我们专注于判断结合律的断言：

```
  -- 起初，我们对结合律一无所知。
```

现在，我们将规则应用到所有已知的判断上来。起始步骤告诉我们对于所有的自然数 `n` 和 `p` 来说，`(zero + n) + p ≡ zero + (n + p)`。归纳步骤告我我们若 `(m + n) + p ≡ m + (n + p)`（在昨天）成

立，那么 `(suc m + n) + p ≡ suc m + (n + p)` （在今天）也成立。我们在今天之前并不知道任何关于结合律的判断，因此此规则并未给出任何新的判断：

```
-- 第一天，我们知道了关于 0 的结合律。
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
```

之后我们重复此过程，因此接下来一天我们知道今天以前的所有判断，以及任何通过此规则添加的判断。起始步骤并未告诉我们新的东西，而如今归归纳步骤添加了更多的判断：

```
-- 第二天，我们知道了关于 0 和 1 的结合律。
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
(1 + 0) + 0 ≡ 1 + (0 + 0)   ...   (1 + 4) + 5 ≡ 1 + (4 + 5)   ...
```

我们再次重复此过程：

```
-- 第三天，我们知道了关于 0、1 和 2 的结合律。
(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)   ...
```

此时规律已经很明显了：

```
-- 第四天，我们知道了关于 0、1、2 和 3 的结合律。
(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)   ...
```

此过程可以继续下去。在第 m 天我们会知道所有第一个数小于 m 的判断。

还有一种完全有限的方法来生成同样的等式，它的证明留作读者的练习。

练习 `finite-|-assoc` （延伸）

请参考前文写出前四天已知的加法结合律的创世故事。

```
-- 请将代码写在此处。
```

用改写来证明结合律

证明可不止一种方法。下面是第二种在 Agda 中证明加法结合律的方法，使用 `rewrite` （改写） 而非等式链：

```
+-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
```

对于起始步骤，我们必须证明：

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

根据加法的起始步骤化简等式两边可得：

```
n + p ≡ n + p
```

此式平凡成立。一个项等于其自身的证明写作 `refl`（自反性）。

对于归纳步骤，我们必须证明：

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

根据加法的归纳步骤化简等式两边可得：

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

在根据归纳假设改写后，这两项相等，其证明同样由 `refl` 给出。根据给定的等式进行改写 可用关键字 `rewrite` 后跟一个该等式的证明来表示。改写不仅可以省去等式链还可以避免 调用`cong`.

使用改写证明交换律

下面是加法交换律的第二个证明，使用 `rewrite` 而非等式链：

```
+-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
```

在最后一行中，用两个等式进行改写被表示为用一条竖线分隔两个相关等式的证明。左边的改写会在右边之前

被执行。

交互式构造证明

看看如何在 Emacs 中用 Agda 的交互式特性来构造另一种结合律的证明会很有启发性。我们从输入以下内容开始:

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

其中的问号表示你想要 Agda 帮你填充的代码。如果你按下 `C-c C-l` （先按 Ctrl-c 再按 Ctrl-l），那么问号会被替换为:

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

空的大括号叫做**洞**（**Hole**），0 是用来指代此洞的编号。洞可能会以绿色高亮显示。 Emacs 还会在屏幕下方创建一个新的窗口并显示文本:

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

这表示 0 号洞需要以所提示的判断的证明来填充。

我们希望对 `m` 进行归纳来证明此命题。将光标移动到洞中并按下`C-c C-c`。它会给出提示:

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

按下 `m` 会拆分该变量，并更新此代码:

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

现在有两个洞了，下方的窗口会告诉你每个洞中需要证明的内容:

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

进入 0 号洞并按下 `C-c C-,` 会显示以下文本:

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

```
p ꞉ ℕ
n ꞉ ℕ
```

它表示在化简之后，0 号洞的目标如上所示，所示类型的变量 `p` 和 `n` 可在证明中使用。 给定目标的证明很平凡，只需进入该目标并按下 `C-c C-r` 即可填充。按下 `C-c C-l` 会将剩下的洞重新编号为 0：

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

进入新的 0 号洞并按下 `C-c C-,` 会显示以下文本：

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

同样，它会给出化简后的目标和可用的变量。在此步骤中，我们需要根据归纳假设进行改写，于是我们来编辑这些文本：

```
+-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
```

进入剩下的洞中并按下 `C-c C-,` 会显示以下文本：

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

给定目标的证明很平凡，只需进入该目标并按下 `C-c C-r` 即可填充并完成证明：

```
+-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
```

练习：`+-swap` （推荐）

请证明对于所有的自然数 `m`、`n` 和 `p`，

```
m + (n + p) ≡ n + (m + p)
```

成立。无需归纳证明，只需应用前面满足结合律和交换律的结果即可。

```
-- 请将代码写在此处。
```

练习 `*-distrib-+`（推荐）

请证明乘法对加法满足分配律，即对于所有的自然数 `m`、`n` 和 `p`，

```
(m + n) * p ≡ m * p + n * p
```

成立。

```
-- 请将代码写在此处。
```

练习 `*-assoc`（推荐）

请证明乘法满足结合律，即对于所有的自然数 `m`、`n` 和 `p`，

```
(m * n) * p ≡ m * (n * p)
```

成立。

```
-- 请将代码写在此处。
```

练习 `*-comm`（实践）

请证明乘法满足交换律，即对于所有的自然数 `m` 和 `n`，

```
m * n ≡ n * m
```

成立。和加法交换律一样，你需要陈述并证明配套的引理。

```
-- 请将代码写在此处。
```

练习 `0∸n≡0`（实践）

请证明对于所有的自然数 `n`，

```
zero ∸ n ≡ zero
```

成立。你的证明需要归纳法吗？

```
-- 请将代码写在此处。
```

练习 `∸-|-assoc`（实践）

请证明饱和减法与加法满足结合律，即对于所有的自然数 `m`、`n` 和 `p`，

```
m ∸ n ∸ p ≡ m ∸ (n + p)
```

成立。

```
-- 请将代码写在此处。
```

练习 `+*^`（延伸）

证明下列三条定律

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

对于所有 `m`、`n` 和 `p` 成立。

练习 `Bin-laws`（延伸）

回想练习 Bin 中定义的一种表示自然数的比特串数据类型 `Bin` 以及要求你定义的函数：

```
inc   ꞉ Bin → Bin
to    ꞉ ℕ → Bin
from  ꞉ Bin → ℕ
```

考虑以下定律，其中 `n` 表示自然数，`b` 表示比特串：

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

对于每一条定律：若它成立，请证明；若不成立，请给出一个反例。

```
-- 请将代码写在此处。
```

标准库

本章中类似的定义可在标准库中找到：

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

Unicode

本章中使用了以下 Unicode：

```
∀  U+2200  对于所有 (\forall, \all)
ʳ  U+02B3  修饰符小写字母 r (\^r)
′  U+2032  撇号 (\')
″  U+2033  双撇号 (\')
‴  U+2034  三撇号 (\')
⁗  U+2057  四撇号 (\')
```

与 `\r` 类似，命令 `\^r` 列出了多种上标右箭头的变体，以及上标的字母 `r`。 命令`\'` 列出了一些撇号（`′ ″ ‴ ⁗`）。

Chapter 3

Relations: 关系的归纳定义

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

在定义了加法和乘法等运算以后，下一步我们来定义**关系（Relation）**，比如说**小于等于**。

导入

```
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)
```

定义关系

小于等于这个关系有无穷个实例，如下所示：

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

但是，我们仍然可以用几行有限的定义来表示所有的实例，如下文所示的一对推理规则：

```
z≤n --------
    zero ≤ n
```

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

以及其 Agda 定义：

```
data _≤_ : ℕ → ℕ → Set where

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

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

在这里，`z≤n` 和 `s≤s`（无空格）是构造子的名称，`zero ≤ n`、`m ≤ n` 和 `suc m ≤ suc n`（带空格）是类型。在这里我们第一次用到了**索引数据类型（Indexed datatype）**。我们使用 `m` 和 `n` 这两个自然数来索引 `m ≤ n` 这个类型。在 Agda 里，由两个及以上短横线开始的行是注释行，我们巧妙利用这一语法特性，用上述形式来表示相应的推理规则。在后文中，我们还会继续使用这一形式。

这两条定义告诉我们相同的两件事：

- **起始步骤**：对于所有的自然数 `n`，命题 `zero ≤ n` 成立。
- **归纳步骤**：对于所有的自然数 `m` 和 `n`，如果命题 `m ≤ n` 成立，那么命题 `suc m ≤ suc n` 成立。

实际上，他们分别给我们更多的信息：

- **起始步骤**：对于所有的自然数 `n`，构造子 `z≤n` 提供了 `zero ≤ n` 成立的证明。
- **归纳步骤**：对于所有的自然数 `m` 和 `n`，构造子 `s≤s` 将 `m ≤ n` 成立的证明 转化为 `suc m ≤ suc n` 成立的证明。

例如，我们在这里以推理规则的形式写出 `2 ≤ 4` 的证明：

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

下面是对应的 Agda 证明：

```
_ : 2 ≤ 4
_ = s≤s (s≤s z≤n)
```

隐式参数

这是我们第一次使用隐式参数。定义不等式时，构造子的定义中使用了 `∀`， 就像我们在下面的命题中使用 `∀` 一样：

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

但是我们这里的定义使用了花括号 `{ }`，而不是小括号 `( )`。这意味着参数是**隐式的（Implicit）**，不需要额外声明。实际上，Agda 的类型检查器 会**推导（Infer）**出它们。因此，我们在 `m + n ≡ n + m` 的证明中需要写出 `+-comm m n`， 在 `zero ≤ n` 的证明中可以省略 `n`。同理，如果 `m≤n` 是 `m ≤ n` 的证明， 那么我们写出 `s≤s m≤n` 作为 `suc m ≤ suc n` 的证明，无需声明 `m` 和 `n`。

如果有希望的话，我们也可以在大括号里显式声明隐式参数。例如，下面是 `2 ≤ 4` 的 Agda 证明，包括了显式声明了的隐式参数：

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

也可以额外加上参数的名字：

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

在后者的形式中，也可以选择只声明一部分隐式参数：

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

但是不可以改变隐式参数的顺序，即便加上了名字。

我们可以写出 `_` 来让 Agda 用相同的推导方式试着推导一个**显式**的项。 例如，我们可以为命题 `+-identityʳ` 定义一个带有隐式参数的变体：

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

我们用 `_` 来让 Agda 从上下文中推导**显式参数**的值。只有 `m` 这一个值能够给出正确的证明，因此 Agda 愉快地填入了它。 如果 Agda 推导值失败，那么它会报一个错误。

优先级

我们如下定义比较的优先级：

```
infix 4 _≤_
```

我们将 `_≤_` 的优先级设置为 4，所以它比优先级为 6 的 `_+_` 结合的更紧，此外， `1 + 2 ≤ 3` 将被解析为 `(1 + 2) ≤ 3`。我们用 `infix` 来表示运算符既不是左结合的， 也不是右结合的。因为 `1 ≤ 2 ≤ 3` 解析为 `(1 ≤ 2) ≤ 3` 或者 `1 ≤ (2 ≤ 3)` 都没有意义。

可决定性

给定两个数，我们可以很直接地决定第一个数是不是小于等于第二个数。我们在此处不给出说明的代码，但我们会在 Decidable 章节重新讨论这个问题。

反演

在我们的定义中，我们从更小的东西得到更大的东西。例如，我们可以从 `m ≤ n` 得出 `suc m ≤ suc n` 的结论，这里的 `suc m` 比 `m` 更大（也就是说，前者包含后者），`suc n` 也比 `n` 更大。但有时我们也 需要从更大的东西得到更小的东西。

只有一种方式能够证明对于任意 `m` 和 `n` 有 `suc m ≤ suc n`。 这让我们能够反演（invert）之前的规则。

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

这里的 `m≤n`（不带空格）是一个变量名，而 `m ≤ n`（带空格）是一个类型， 且后者是前者的类型。在 Agda 中，将类型中的空格去掉来作为变量名是一种常见的约定。

并不是所有规则都可以反演。实际上，`z≤n` 的规则没有非隐式的假设，因此它没有可以被反演的规则。但这种反演通常是成立的。

反演的另一个例子是证明只存在一种情况使得一个数字能够小于或等于零。

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

```
inv-≤n ≤n = refl
```

序关系的性质

数学家对于关系的常见性质给出了约定的名称。

- **自反（Reflexive）**：对于所有的 `n`，关系 `n ≤ n` 成立。
- **传递（Transitive）**：对于所有的 `m`、`n` 和 `p`，如果 `m ≤ n` 和 `n ≤ p` 成立，那么 `m ≤ p` 也成立。
- **反对称（Anti-symmetric）**：对于所有的 `m` 和 `n`，如果 `m ≤ n` 和 `n ≤ m` 同时成立，那么 `m ≡ n` 成立。
- **完全（Total）**：对于所有的 `m` 和 `n`，`m ≤ n` 或者 `n ≤ m` 成立。

`_≤_` 关系满足上述四条性质。

对于上述性质的组合也有约定的名称。

- **预序（Preorder）**：满足自反和传递的关系。
- **偏序（Partial Order）**：满足反对称的预序。
- **全序（Total Order）**：满足完全的偏序。

如果你进入了关于关系的聚会，你现在知道怎么样和人讨论了，可以讨论关于自反、传递、反对称和完全，或者问一问这是不是预序、偏序或者全序。

更认真的来说，如果你在阅读论文时碰到了一个关系，本文的介绍让你可以对关系有基本的了解和判断，来判断这个关系是不是预序、偏序或者全序。一个认真的作者一般会在文章指出这个关系具有（或者缺少）上述性质，比如说指出新定义的关系是一个偏序而不是全序。

练习 `orderings`（实践）

给出一个不是偏序的预序的例子。

```
-- 请将代码写在此处。
```

给出一个不是全序的偏序的例子。

```
-- 请将代码写在此处。
```

自反性

我们第一个来证明的性质是自反性：对于任意自然数 `n`，关系 `n ≤ n` 成立。我们使用标准库的惯例来隐式申明参数，在使用自反性的证明时这样可以更加方便。

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

这个证明直接在 `n` 上进行归纳。在起始步骤中，`zero ≤ zero` 由 `z≤n` 证明；在归纳步骤中， 归纳假设 `≤-refl {n}` 给我们带来了 `n ≤ n` 的证明，我们只需要使用 `s≤s`，就可以获得 `suc n ≤ suc n` 的证明。

在 Emacs 中来交互式地证明自反性是一个很好的练习，可以使用洞，以及 `C-c C-c`、`C-c C-,` 和 `C-c C-r` 命令。

传递性

我们第二个证明的性质是传递性：对于任意自然数 `m` 和 `n`，如果 `m ≤ n` 和 `n ≤ p` 成立，那么 `m ≤ p` 成立。同样，`m`、`n` 和 `p` 是隐式参数：

```
≤-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)
```

这里我们在 `m ≤ n` 的**证据（Evidence）**上进行归纳。在起始步骤里，第一个不等式因为 `z≤n` 而成立， 那么结论亦可由 `z≤n` 而得出。在这里，`n ≤ p` 的证明是不需要的，我们用 `_` 来表示这个 证明没有被使用。

在归纳步骤中，第一个不等式因为 `s≤s m≤n` 而成立，第二个不等式因为 `s≤s n≤p` 而成立， 所以我们已知 `suc m ≤ suc n` 和 `suc n ≤ suc p`，求证 `suc m ≤ suc p`。 通过归纳假设 `≤-trans m≤n n≤p`，我们得知 `m ≤ p`，在此之上使用 `s≤s` 即可证。

`≤-trans (s≤s m≤n) z≤n` 不可能发生，因为第一个不等式告诉我们中间的数是一个 `suc n`，而第二个不等式告诉我们中间的数是 `zero`。Agda 可以推断这样的情况不可能发现，所以我们不需要 （也不可以）列出这种情况。

我们也可以将隐式参数显式地声明。

```
≤-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)
```

有人说这样的证明更加的清晰，也有人说这个更长的证明让人难以抓住证明的重点。我们一般选择使用简短的证明。

对于性质成立证明进行的归纳（如上文中对于 `m ≤ n` 的证明进行归纳），相比于对于性质成立的值进行的归纳（如对于 `m` 进行归纳），有非常大的价值。我们会经常使用这样的方法。

同样，在 Emacs 中来交互式地证明传递性是一个很好的练习，可以使用洞，以及`C-c C-c`、 `C-c C-,` 和 `C-c C-r` 命令。

反对称性

我们证明的第三个性质是反对称性：对于所有的自然数 `m` 和`n`，如果 `m ≤ n` 和 `n ≤ m` 同时成立，那么 `m ≡ n` 成立：

```
≤-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)
```

同样，我们对于 `m ≤ n` 和 `n ≤ m` 的证明进行归纳。

在起始步骤中，两个不等式都因为 `z≤n` 而成立。因此我们已知`zero ≤ zero` 和 `zero ≤ zero`， 求证 `zero ≡ zero`，由自反性可证。（注：由等式的自反性可证，而不是不等式的自反性）

在归纳步骤中，第一个不等式因为 `s≤s m≤n` 而成立，第二个等式因为`s≤s n≤m` 而成立。因此我们已知 `suc m ≤ suc n` 和`suc n ≤ suc m`，求证 `suc m ≡ suc n`。归纳假设`≤-antisym m≤n n≤m` 可以证明 `m ≡ n`，因此我们可以使用同余性完成证明。

练习`≤-antisym-cases`（实践）

上面的证明中省略了一个参数是 `z≤n`，另一个参数是 `s≤s` 的情况。为什么可以省略这种情况？

```
-- 请将代码写在此处。
```

完全性

我们证明的第四个性质是完全性：对于任何自然数 `m` 和 `n`，`m ≤ n` 或者 `n ≤ m` 成立。在 `m` 和 `n` 相等时，两者同时成立。

我们首先来说明怎么样不等式才是完全的：

```
data Total (m n : ℕ) : Set where

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

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

`Total m n` 成立的证明有两种形式：`forward m≤n` 或者 `flipped n≤m`，其中 `m≤n` 和 `n≤m` 分别是 `m ≤ n` 和 `n ≤ m` 的证明。

（如果你对于逻辑学有所了解，上面的定义可以由析取（Disjunction）表示。我们会在 Connectives 章节介绍析取。）

这是我们第一次使用带*参数*的数据类型，这里 `m` 和 `n` 是参数。这等同于下面的索引数据类型：

```
data Total′ : ℕ → ℕ → Set where

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

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

类型里的每个参数都转换成构造子的一个隐式参数。索引数据类型中的索引可以变化，正如在 `zero ≤ n` 和 `suc m ≤ suc n` 中那样，而参数化数据类型不一样，其参数必须保持相同，正如在 `Total m n` 中那样。参数化的声明更短，更易于阅读，而且有时可以帮助到 Agda 的终结检查器，所以我们尽可能地使用它们，而不

是索引数据类型。

在上述准备工作完成后，我们定义并证明完全性。

```
≤-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)
```

这里，我们的证明在两个参数上进行归纳，并按照情况分析：

- **第一起始步骤**：如果第一个参数是 `zero`，第二个参数是 `n`，那么 forward 条件成立，我们使用 `z≤n` 作为 `zero ≤ n` 的证明。
- **第二起始步骤**：如果第一个参数是 `suc m`，第二个参数是 `zero`，那么 flipped 条件成立，我们使用 `z≤n` 作为 `zero ≤ suc m` 的证明。
- **归纳步骤**：如果第一个参数是 `suc m`，第二个参数是 `suc n`，那么归纳假设 `≤-total m n` 可以给出如下推断：
  - 归纳假设的 forward 条件成立，以 `m≤n` 作为 `m ≤ n` 的证明。以此我们可以使用 `s≤s m≤n` 作为 `suc m ≤ suc n` 来证明 forward 条件成立。
  - 归纳假设的 flipped 条件成立，以 `n≤m` 作为 `n ≤ m` 的证明。以此我们可以使用 `s≤s n≤m` 作为 `suc n ≤ suc m` 来证明 flipped 条件成立。

这是我们第一次在 Agda 中使用 `with` 语句。`with` 关键字后面有一个表达式和一或多行。每行以省略号（`...`）和一个竖线（`|`）开头，后面跟着用来匹配表达式的模式，和等式的右手边。

使用 `with` 语句等同于定义一个辅助函数。比如说，上面的定义和下面的等价：

```
≤-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)
```

这也是我们第一次在 Agda 中使用 `where` 语句。`where` 关键字后面有一或多条定义，其必须被缩进。之前等式左手边的约束变量（此例中的 `m` 和 `n`）在嵌套的定义中仍然在作用域内。在嵌套定义中的约束标识符（此例中的 `helper`）在等式的右手边的作用域内。

如果两个参数相同，那么两个情况同时成立，我们可以返回任一证明。上面的代码中我们返回 forward 条件，但是我们也可以返回 flipped 条件，如下：

```
≤-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)
```

两者的区别在于上述代码在对于第一个参数进行模式匹配之前先对于第二个参数先进行模式匹配。

单调性

如果在聚会中碰到了一个运算符和一个序，那么有人可能会问这个运算符对于这个序是不是**单调的**(**Monotonic**)。比如说，加法对于小于等于是单调的，这意味着：

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

这个证明可以用我们学会的方法，很直接的来完成。我们最好把它分成三个部分，首先我们证明加法对于小于等于在右手边是单调的：

```
+-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)
```

我们对于第一个参数进行归纳。

- **起始步骤**：第一个参数是 `zero`，那么 `zero + p ≤ zero + q` 可以化简为 `p ≤ q`，其证明由 `p≤q` 给出。
- **归纳步骤**：第一个参数是 `suc n`，那么 `suc n + p ≤ suc n + q` 可以化简为 `suc (n + p) ≤ suc (n + q)`。归纳假设 `+-monoʳ-≤ n p q p≤q` 可以证明 `n + p ≤ n + q`，我们在此之上使用 `s≤s` 即可得证。

接下来，我们证明加法对于小于等于在左手边是单调的。我们可以用之前的结论和加法的交换律来证明：

```
+-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
```

用 `+-comm m p` 和 `+-comm n p` 来重写，可以让 `m + p ≤ n + p` 转换成 `p + n ≤ p + m`，而我们可以用 `+-monoʳ-≤ p m n m≤n` 来证明。

最后，我们把前两步的结论结合起来：

```
+-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)
```

使用 `+-monoˡ-≤ m n p m≤n` 可以证明 `m + p ≤ n + p`，使用 `+-monoʳ-≤ n p q p≤q` 可以证明 `n + p ≤ n + q`，用传递性把两者连接起来，我们可以获得 `m + p ≤ n + q` 的证明，如上所示。

练习 `*-mono-≤`（延伸）

证明乘法对于小于等于是单调的。

```
-- 请将代码写在此处。
```

严格不等关系

我们可以用类似于定义不等关系的方法来定义严格不等关系。

```
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
```

严格不等关系与不等关系最重要的区别在于，0 小于任何数的后继，而不小于 0。

显然，严格不等关系不是自反的，而是**非自反的（Irreflexive）**，表示 `n < n` 对于任何值 `n` 都不成立。和不等关系一样，严格不等关系是传递的。严格不等关系不是完全的，但是满足一个相似的性质：三分律

(Trichotomy)：对于任意的 `m` 和 `n`，`m < n`、`m ≡ n` 或者 `m > n` 三者有且仅有一者成立。(我们定义 `m > n` 当且仅当 `n < m` 成立时成立) 严格不等关系对于加法和乘法也是单调的。

我们把一部分上述性质作为习题。非自反性需要逻辑非，三分律需要证明三者是互斥的，因此这两个性质暂不做为习题。我们会在 Negation 章节来重新讨论。

我们可以直接地来证明 `suc m ≤ n` 蕴涵了 `m < n`，及其逆命题。因此我们亦可从不等关系的性质中，使用此性质来证明严格不等关系的性质。

练习 `<-trans` (推荐)

证明严格不等是传递的。

```
-- 请将代码写在此处。
```

练习 `trichotomy` (实践)

证明严格不等关系满足弱化的三元律，证明对于任意 `m` 和 `n`，下列命题有一条成立：

- `m < n`，
- `m ≡ n`，或者
- `m > n`。

定义 `m > n` 为 `n < m`。你需要一个合适的数据类型声明，如同我们在证明完全性中使用的那样。(我们会在介绍完否定之后证明三者是互斥的。)

```
-- 请将代码写在此处。
```

练习 `+-mono-<` (实践)

证明加法对于严格不等关系是单调的。正如不等关系中那样，你可以需要额外的定义。

```
-- 请将代码写在此处。
```

练习 `≤-iff-<` (推荐)

证明 `suc m ≤ n` 蕴涵了 `m < n`，及其逆命题。

```
-- 请将代码写在此处。
```

练习 `<-trans-revisited`（实践）

用另外一种方法证明严格不等是传递的，使用之前证明的不等关系和严格不等关系的联系，以及不等关系的传递性。

```
-- 请将代码写在此处。
```

奇和偶

作为一个额外的例子，我们来定义奇数和偶数。不等关系和严格不等关系是**二元关系**，而奇偶性是**一元关系**，有时也被叫做**谓词（Predicate）**：

```
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)
```

一个数是偶数，如果它是 0，或者是奇数的后继。一个数是奇数，如果它是偶数的后继。

这是我们第一次定义一个相互递归的数据类型。因为每个标识符必须在使用前声明，所以我们首先声明索引数据类型 `even` 和 `odd`（省略 `where` 关键字和其构造子的定义），然后声明其构造子（省略其签名 `ℕ → Set`，因为在之前已经给出）。

这也是我们第一次使用**重载（Overloaded）**的构造子。这意味着不同类型的构造子拥有相同的名字。在这里 `suc` 表示下面三种构造子其中之一：

```
suc : ℕ → ℕ

suc : ∀ {n : ℕ}
```

```
  → odd n
    ------------
  → even (suc n)

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

同理，`zero` 表示两种构造子的一种。因为类型推导的限制，Agda 不允许重载已定义的名字，但是允许重载构造子。我们推荐将重载限制在有关联的定义中，如我们所做的这样，但这不是必须的。

我们证明两个偶数之和是偶数：

```
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)
```

与相互递归的定义对应，我们用两个相互递归的函数，一个证明两个偶数之和是偶数，另一个证明一个奇数与一个偶数之和是奇数。

这是我们第一次使用相互递归的函数。因为每个标识符必须在使用前声明，我们先给出两个函数的签名，然后再给出其定义。

要证明两个偶数之和为偶，我们考虑第一个数为偶数的证明。如果是因为第一个数为 0，那么第二个数为偶数的证明即为和为偶数的证明。如果是因为第一个数为奇数的后继，那么和为偶数是因为他是一个奇数和一个偶数的和的后续，而这个和是一个奇数。

要证明一个奇数和一个偶数的和是奇数，我们考虑第一个数是奇数的证明。如果是因为它是一个偶数的后继，那么和为奇数，因为它是两个偶数之和的后继，而这个和是一个偶数。

练习 `o+o≡e`（延伸）

证明两个奇数之和为偶数。

```
-- 请将代码写在此处。
```

练习 Bin-predicates（延伸）

回忆我们在练习 Bin 中定义了一个数据类型 `Bin` 来用二进制字符串表示自然数。这个表达方法不是唯一的，因为我们在开头加任意个 0。因此，11 可以由以下方法表示：

```
⟨⟩ I O I I
⟨⟩ O O I O I I
```

定义一个谓词

```
Can : Bin → Set
```

其在一个二进制字符串的表示是标准的（Canonical）时成立，表示它没有开头的 0。在两个 11 的表达方式中，第一个是标准的，而第二个不是。在定义这个谓词时，你需要一个辅助谓词：

```
One : Bin → Set
```

其仅在一个二进制字符串开头为 1 时成立。一个二进制字符串是标准的，如果它开头是 1（表示一个正数），或者它仅是一个 0 （表示 0）。

证明递增可以保持标准性。

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

证明从自然数转换成的二进制字符串是标准的。

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

证明将一个标准的二进制字符串转换成自然数之后，再转换回二进制字符串与原二进制字符串相同。

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

（提示：对于每一条习题，先从 `One` 的性质开始。此外，你或许还需要证明若 `One b` 成立，则 `1` 小于或等于 `from b` 的结果。）

```
-- 请将代码写在此处。
```

标准库

标准库中有类似于本章介绍的定义：

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

在标准库中，`≤-total` 是使用析取定义的（我们将在 Connectives 章节定义）。`+-monoʳ-≤`、`+-monoˡ-≤` 和 `+-mono-≤` 的证明方法和本书不同。 更多的参数是隐式申明的。

Unicode

本章使用了如下 Unicode 符号：

```
≤  U+2264  小于等于 (\<=, \le)
≥  U+2265  大于等于 (\>=, \ge)
ˡ  U+02E1  小写字母 L 标识符 (\^l)
ʳ  U+02B3  小写字母 R 标识符 (\^r)
```

`\^l` 和 `\^r` 命令给出了左右箭头，以及上标字母 `l` 和 `r`。

Chapter 4

Equality：相等性与等式推理

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

我们在论证的过程中经常会使用相等性。给定两个都为 `A` 类型的项 `M` 和 `N`，我们用 `M ≡ N` 来表示 `M` 和 `N` 可以相互替换。在此之前，我们将相等性作为一个基础运算，而现在我们来说明如果将其定义为一个归纳的数据类型。

导入

本章节没有导入的内容。本书的每一章节，以及 Agda 标准库的每个模块都导入了相等性。我们在此定义相等性，导入其他内容将会产生冲突。

相等性

我们如下定义相等性：

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

用其他的话来说，对于任意类型 `A` 和任意 `A` 类型的 `x`，构造子 `refl` 提供了 `x ≡ x` 的证明。所以，每个值等同于它本身，我们并没有其他办法来证明值的相等性。这个定义里有不对称的地方，`_≡_` 的第一个参数（Argument）由 `x : A` 给出，而第二个参数（Argument）则是由 `A → Set` 的索引给出。这和我们尽可能多的使用参数（Parameter）的理念相符。`_≡_` 的第一个参数（Argument）可以作为一个参数（Parameter），因为它不会变，而第二个参数（Argument）则必须是一个索引，这样它才可以等用于第一个。

我们如下定义相等性的优先级：

```
infix 4 _≡_
```

我们将 `_≡_` 的优先级设置为 4，与 `_≤_` 相同，所以其它算术运算符的结合都比它紧密。由于它既不是左结合，也不是右结合的，因此 `x ≡ y ≡ z` 是不合法的。

相等性是一个等价关系（Equivalence Relation）

一个等价关系是自反、对称和传递的。其中自反性可以通过构造子 `refl` 直接从相等性的定义中得来。 我们可以直接地证明其对称性：

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

这个证明是怎么运作的呢？`sym` 参数的类型是 `x ≡ y`，但是等式的左手边被 `refl` 模式实例化了，这要求 `x` 和 `y` 相等。因此，等式的右手边需要一个类型为 `x ≡ x` 的项，用 `refl` 即可。

交互式地证明 `sym` 很有教育意义。首先，我们在左手边使用一个变量来表示参数，在右手边使用一个洞：

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

如果我们进入这个洞，使用 `C-c C-,`，Agda 会告诉我们：

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

在这个洞里，我们使用 `C-c C-c e`，Agda 会将 `e` 逐一展开为所有可能的构造子。 此处只有一个构造子：

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

```
  → y ≡ x
sym refl = {! !}
```

如果我们再次进入这个洞，重新使用 `C-c C-,`，然后 Agda 现在会告诉我们：

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

这是一个重要的步骤—— Agda 发现了 `x` 和 `y` 必须相等，才能与模式 `refl` 相匹配。

最后，我们回到洞里，使用 `C-c C-r`，Agda 将会把洞变成一个可以满足给定类型的构造子实例。

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

我们至此完成了与之前给出证明相同的证明。

传递性亦是很直接：

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

同样，交互式地证明这个特性是一个很好的练习，尤其是观察 Agda 的已知内容根据参数的实例而变化的过程。

合同性和替换性

相等性满足合同性（Congruence）。如果两个项相等，那么对它们使用相同的函数，其结果仍然相等：

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

两个参数的函数也满足合同性：

```
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
```

在函数上的等价性也满足合同性。如果两个函数是相等的，那么它们作用在同一项上的结果是相等的：

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

相等性也满足替换性（Substitution）。如果两个值相等，其中一个满足某谓词，那么另一个也满足此谓词。

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

等式串

我们在此演示如何使用等式串来论证，正如本书中使用证明形式。我们讲声明放在一个叫做 `≡-Reasoning` 的模块里，与 Agda 标准库中的格式相对应。

```
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
```

这是我们第一次使用嵌套的模块。它包括了关键字 `module` 和后续的模块名、隐式或显式参数， 关键字 `where`，和模块中的内容（在缩进内）。模块里可以包括任何形式的声明，也可以包括其他模块。嵌套的模块和本书每章节所定义的顶层模块相似，只是顶层模块不需要缩进。打开（Open）一个模块会把模块内的所有定义导入进当前的环境中。

举个例子，我们来看看如何用等式串证明传递性：

```
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
  ∎
```

根据其定义，等式右边会被解析成如下：

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

这里 `begin` 的使用纯粹是装饰性的，因为它直接返回了其参数。其参数包括了 `_≡⟨_⟩_` 作用于 `x`、`x≡y` 和 `y ≡⟨ y≡z ⟩ (z ∎)`。第一个参数是一个项 `x`，而第二、第三个参数分别是等式 `x ≡ y`、`y ≡ z` 的证明，

它们在 `_≡⟨_⟩_` 的定义中用 `trans` 连接起来，形成 `x ≡ z` 的证明。`y ≡ z` 的证明包括了 `_≡⟨_⟩_` 作用于 `y`、`y≡z` 和 `z ∎`。第一个参数是一个项 `y`，而第二、第三个参数分别是等式 `y ≡ z`、`z ≡ z` 的证明，它们在 `_≡⟨_⟩_` 的定义中用 `trans` 连接起来，形成 `y ≡ z` 的证明。最后，`z ≡ z` 的证明包括了 `_∎` 作用于 `z` 之上，使用了 `refl`。经过化简，上述定义等同于：

```
trans x≡y (trans y≡z refl)
```

我们可以把任意等式串转化成一系列的 `trans` 的使用。这样的证明更加精简，但是更难以阅读。`∎` 的小窍门意味着等式串化简成为的一系列 `trans` 会以 `trans e refl` 结尾，尽管只需要 `e` 就足够了，这里的 `e` 是等式的证明。

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
```

等式串的另外一个例子

我们重新证明加法的交换律来作为等式串的第二个例子。我们首先重复自然数和加法的定义。我们不能导入它们（正如本章节开头中所解释的那样），因为那样会产生一个冲突：

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

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

为了节约空间，我们假设两条引理（而不是证明它们）：

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

这是我们第一次使用假设（Postulate）。假设为一个标识符指定一个签名，但是不提供定义。我们在这里假设之前证明过的东西，来节约空间。假设在使用时必须加以注意。如果假设的内容为假，那么我们可以证明出任何东西。

我们接下来重复交换律的证明：

```
+-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
  ∎
```

论证的过程和之前的相似。我们在不需要解释的地方使用 `_≡⟨⟩_`，我们可以认为 `_≡⟨⟩_` 和 `_≡⟨ refl ⟩_` 是等价的。

Agda 总是认为一个项与其化简的项是等价的。我们之所以可以写出

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

是因为 Agda 认为它们是一样的。这也意味着我们可以交换两行的顺序，写出

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

而 Agda 并不会反对。Agda 只会检查由 `≡⟨⟩` 隔开的项是否化简后相同。 而书写的顺序合不合理则是由我们自行决定。

练习 `≤-Reasoning`（延伸）

Relations 章节中的单调性证明亦可以用相似于 `≡-Reasoning` 的，更易于理解的形式给出。 相似地来定义 `≤-Reasoning`，并用其重新给出加法对于不等式是单调的证明。重写 `+-monoˡ-≤`、`+-monoʳ-≤` 和 `+-mono-≤`。

```
-- 请将代码写在此处。
```

重写

考虑一个自然数的性质，比如说一个数是偶数。我们重复之前给出的定义：

```
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)
```

在前面的部分中，我们证明了加法满足交换律。给定 `even (m + n)` 成立的证据，我们应当可以用它来做 `even (n + m)` 成立的证据。

Agda 对这种论证有特殊记法的支持——我们之前提到过的 `rewrite` 记法。来启用这种记法， 我们只用编译程序指令来告诉 Agda 什么类型对应相等性：

```
{-# BUILTIN EQUALITY _≡_ #-}
```

我们然后就可以如下证明求证的性质：

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

在这里，`ev` 包括了所有 `even (m + n)` 成立的证据，我们证明它亦可作为 `even (n + m)` 成立的证据。一般来说，关键字 `rewrite` 之后跟着一个等式的证明，这个等式被用于重写目标和任意作用域内变量的类型。

交互性地证明 `even-comm` 是很有帮助的。一开始，我们先给左边的参数赋予变量，给右手边放上一个洞：

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

如果我们进入洞里，输入 `C-c C-,`，Agda 会报告：

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

现在我们加入重写：

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

如果我们再次进入洞里，并输入 `C-c C-,`，Agda 现在会报告：

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

目标里的参数被交换了。现在 `ev` 显然满足目标条件，输入`C-c C-a` 会用 `ev` 来填充这个洞。 命令 `C-c C-a` 可以进行自动搜索，检查作用域内的变量是否和目标有相同的类型。

多重重写

我们可以多次使用重写，以竖线隔开。举个例子，这里是加法交换律的第二个证明，使用重写而不是等式串：

```
+-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
```

这个证明更加的简短。之前的证明用 `cong suc (+-comm m n)` 作为使用归纳假设的说明， 而这里我们使用 `+-comm m n` 来重写就足够了，因为重写可以将合同性考虑在其中。尽管使用重写的证明更加的简短，使用等式串的证明能容易理解，我们将尽可能的使用后者。

深入重写

`rewrite` 记法实际上是 `with` 抽象的一种应用：

```
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
```

总的来着，我们可以在 `with` 后面跟上任何数量的表达式，用竖线分隔开，并且在每个等式中使用相同个数的模式。我们经常将表达式和模式如上对齐。这个第一列表明了 `m + n` 和 `n + m` 是相同的，第二列使用相应等式来证明的前述的断言。注意在这里使用的点模式（Dot Pattern），`.(n + m)`。点模式由一个点和一个表达式组成，在其他信息迫使这个值和点模式中的值相等时使用。在这里，`m + n` 和 `n + m` 由后续的 `+-comm m n` 与 `refl` 的匹配来识别。我们可能会认为第一种情况是多余的，因为第二种情况中才蕴涵了需要的信息。但实际上 Agda 在这件事上很挑剔——省略第一条或者更换顺序会让 Agda 报告一个错误。（试一试你就知道!）

在这种情况中，我们也可以使用之前定义的替换函数来避免使用重写：

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

尽管如此，重写是 Agda 工具箱中很重要的一部分。我们会偶尔使用它，但是它有的时候是必要的。

莱布尼兹（Leibniz）相等性

我们使用的相等性断言的形式源于 Martin-Löf，于 1975 年发表。一个更早的形式源于莱布尼兹， 于 1686 年发表。莱布尼兹断言的相等性表示不可分辨的实体（Identity of Indiscernibles）：两个对象相等当且仅当它们满足完全相同的性质。这条原理有时被称作莱布尼兹定律（Leibniz' Law），与史波克定律紧密相关：“一个不造成区别的区别不是区别”。我们在这里定义莱布尼兹相等性，并证明两个项满足莱布尼兹相等性当且仅当其满足 Martin-Löf 相等性。

莱布尼兹不等式一般如下来定义：`x ≐ y` 当每个对于 `x` 成立的性质 `P` 对于 `y` 也成立时成立。可能这有些出乎意料，但是这个定义亦足够保证其相反的命题：每个对于 `y` 成立的性质 `P` 对于 `x` 也成立。

令 `x` 和 `y` 为类型 `A` 的对象。我们定义 `x ≐ y` 成立，当每个对于类型 `A` 成立的谓词 `P`，我们有 `P x` 蕴涵了 `P y`：

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

我们不能在左手边使用 `x ≐ y`，取而代之我们使用 `_≐_ {A} x y` 来提供隐式参数 `A`，这样 `A` 可以出现在右手边。

这是我们第一次使用等级（Levels）。我们不能将 `Set` 赋予类型 `Set`，因为这会导致自相矛盾， 比如罗素悖论（Russell's Paradox）或者 Girard 悖论。不同的是，我们有一个阶级的类型：其中 `Set : Set₁`，`Set₁ : Set₂`，以此类推。实际上，`Set` 本身就是 `Set₀` 的缩写。定义 `_≐_` 的等式在右手边提到了 `Set`，因此签名中必须使用 `Set₁`。我们稍后将进一步介绍等级。

莱布尼兹相等性是自反和传递的。自反性由恒等函数的变种得来，传递性由函数组合的变种得来：

```
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)
```

对称性就没有那么显然了。我们需要证明如果对于所有谓词 `P`，`P x` 蕴涵 `P y`，那么反方向的蕴涵也成立。

```
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
```

给定 `x ≐ y` 和一个特定的 `P`，我们需要构造一个 `P y` 蕴涵 `P x` 的证明。 我们首先用一个谓词 `Q` 将相等性实例化，使得 `Q z` 在 `P z` 蕴涵 `P x` 时成立。 `Q x` 这个性质是显然的，由自反性可以得出，由此通过 `x ≐ y` 就能推出 `Q y` 成立。而 `Q y` 正是我们需要的证明，即 `P y` 蕴涵 `P x`。

我们现在来证明 Martin-Löf 相等性蕴涵了莱布尼兹相等性，以及其逆命题。在正方向上， 如果我们已知 `x ≡ y`，我们需要对于任意的 `P`，将 `P x` 的证明转换为 `P y` 的证明。 我们很容易就可以做到这一点，因为 `x` 与 `y` 相等意味着任何 `P x` 的证明即是 `P y` 的证明。

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

因为这个方向由替换性可以得来，如之前证明的那样。

在反方向上，我们已知对于任何 `P`，我们可以将 `P x` 的证明转换成 `P y` 的证明， 我们需要证明 `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
```

此证明与莱布尼兹相等性的对称性证明相似。我们取谓词 `Q`，使得 `Q z` 在 `x ≡ z` 成立时成立。 那么 `Q x` 是显然的，由 Martin Löf 相等性的自反性得来。从而 `Q y` 由 `x ≐ y` 可得，而 `Q y` 即是我们所需要的 `x ≡ y` 的证明。

（本部分的内容由此处改编得来：*≐≃≡: Leibniz Equality is Isomorphic to Martin-Löf Identity, Parametrically* 作者：Andreas Abel、Jesper Cockx、Dominique Devries、Andreas Nuyts 与 Philip Wadler，草稿，2017）

全体多态

正如我们之前看到的那样，不是每个类型都属于 `Set`，但是每个类型都属于类型阶级的某处，`Set₀`、`Set₁`、`Set₂` 等等。其中 `Set` 是 `Set₀` 的缩写，此外 `Set₀ : Set₁`，`Set₁ : Set₂`，以此类推。当我们需要比较两个属于 `Set` 的类型的值时，我们之前给出的定义是足够的，但如果我们需要比较对于任何等级 `ℓ`，两个属于 `Set ℓ` 的类型的值该怎么办呢？

答案是全体多态（Universe Polymorphism），一个定义可以根据任何等级 `ℓ` 来做出。 为了使用等级，我们首先导入下列内容：

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

我们将构造子 `zero` 和 `suc` 重命名至 `lzero` 和 `lsuc`，为了防止自然数和等级之间的混淆。

等级与自然数是同构的，有相似的构造子：

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

`Set₀`、`Set₁`、`Set₂` 等名称，是下列的简写：

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

以此类推。我们还有一个运算符：

```
_⊔_ : Level → Level → Level
```

给定两个等级，返回两者中较大的那个。

下面是相等性的定义，推广到任意等级：

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

相似的，下面是对称性的推广定义：

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

为了简介，我们在本书中给出的定义将避免使用全体多态，但是大多数标准库中的定义，包括相等性的定义，都推广到了任意等级，如上所示。

下面是莱布尼兹相等性的推广定义：

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

之前，签名中使用了 `Set₁` 来作为一个值包括了 `Set` 的类型；而此处，我们使用 `Set (lsuc ℓ)` 来作为一

个值包括了 `Set ℓ` 的类型。

标准库中的大部分函数都泛化到了任意层级。例如，以下是复合的定义。

```
_∘_ : ∀ {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)
```

更多关于层级的信息可以从 Agda 文档中查询。

标准库

标准库中可以找到与本章节中相似的定义。Agda 标准库将 `_≡⟨_⟩_` 定义为 `step-≡`，它反转了参数的顺序。标准库还定义了一个语法宏，它可以在你导入 `step-≡` 时被自动导入，它能够恢复原始的参数顺序：

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

这里的导入以注释的形式给出，以防止冲突，如引言中解释的那样。

Unicode

本章节使用下列 Unicode：

```
≡  U+2261  等同于 (\==, \equiv)
⟨  U+27E8  数学左尖括号 (\<)
⟩  U+27E9  数学右尖括号 (\>)
∎  U+220E  证毕 (\qed)
≐  U+2250  趋近于极限 (\.=)
ℓ  U+2113  手写小写 L (\ell)
⊔  U+2294  正方形向上开口 (\lub)
₀  U+2080  下标 0 (\_0)
₁  U+2081  下标 1 (\_1)
₂  U+2082  下标 2 (\_2)
```

Chapter 5

Isomorphism: 同构与嵌入

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

本部分介绍同构（Isomorphism）与嵌入（Embedding）。同构可以断言两个类型是相等的，嵌入可以断言一个类型比另一个类型小。我们会在下一章中使用同构来展示类型上的运算，例如积或者和，满足类似于交换律、结合律和分配律的性质。

导入

```
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 表达式

本章节开头将补充一些有用的基础知识：lambda 表达式，函数组合，以及外延性。

Lambda 表达式提供了一种简洁的定义函数的方法，且不需要提供函数名。一个如同这样的项：

```
λ{ P₁ → N₁; ⋯ ; Pₙ → Nₙ }
```

等同于定义一个函数 `f`，使用下列等式：

```
f P₁ = N₁
⋯
f Pₙ = Nₙ
```

其中 `Pₙ` 是模式（即等式的左手边），`Nₙ` 是表达式（即等式的右手边）。

如果只有一个等式，且模式是一个变量，我们亦可使用下面的语法：

```
λ x → N
```

或者

```
λ (x : A) → N
```

两个都与 `λ{x → N}` 等价。后者可以指定函数的作用域。

往往使用匿名的 lambda 表达式比使用带名字的函数要方便：它避免了冗长的类型声明；其定义出现在其使用的地方，所以在书写时不需要记得提前声明，在阅读时不需要上下搜索函数定义。

函数组合 （Function Composition）

接下来，我们将使用函数组合：

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

`g ∘ f` 是一个函数，先使用函数 `f`，再使用函数 `g`。 一个等价的定义，使用 lambda 表达式，如下：

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

外延性（Extensionality）

外延性断言了区分函数的唯一方法是应用它们。如果两个函数作用在相同的参数上永远返回相同的结果，那么两个函数相同。这是 `cong-app` 的逆命题，在之前有所介绍。

Agda 并不预设外延性，但我们可以假设其成立：

```
postulate
  extensionality : ∀ {A B : Set} {f g : A → B}
```

```
    → (∀ (x ∶ A) → f x ≡ g x)
      ----------------------
    → f ≡ g
```

假设外延性不会造成困顿，因为我们知道它与 Agda 使用的理论是连贯一致的。

举个例子，我们考虑两个库都定义了加法，一个按照我们在 Naturals 章节中那样定义，另一个如下，反过来定义：

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

通过使用交换律，我们可以简单地证明两个运算符在给定相同参数的情况下，会返回相同的值：

```
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)
```

然而，有时断言两个运算符是无法区分的会更加方便。我们可以使用两次外延性：

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

我们偶尔需要在之后的情况中假设外延性。

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.

同构（Isomorphism）

如果两个集合有一一对应的关系，那么它们是同构的。 下面是同构的正式定义：

```
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 _≃_
```

我们来一一展开这个定义。一个集合 `A` 和 `B` 之间的同构有四个要素： + 从 `A` 到 `B` 的函数 `to` + 从 `B` 回到 `A` 的函数 `from` + `from` 是 `to` 的*左逆*（left-inverse）的证明 `from∘to` + `from` 是 `to` 的*右逆*（right-inverse）的证明 `to∘from`

具体来说，第三条断言了 `from ∘ to` 是恒等函数，第四条断言了 `to ∘ from` 是恒等函数， 它们的名称由此得来。声明 `open _≃_` 使得 `to`、`from`、`from∘to` 和 `to∘from` 在当前作用域内可用，否则我们需要使用类似 `_≃_.to` 的写法。

这是我们第一次使用**记录（Record）**。记录声明的行为类似于一个单构造子的数据声明（二者稍微有些不同，我们会在 Connectives 一章中讨论）：

```
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
```

我们用下面的语法来构造一个记录类型的值：

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

这与使用相应的归纳类型的构造子对应：

```
mk-≃′ f g g∘f f∘g
```

其中 `f`、`g`、`g∘f` 和 `f∘g` 是相应类型的值。

同构是一个等价关系

同构是一个等价关系。这意味着它自反、对称、传递。要证明同构是自反的，我们用恒等函数作为 `to` 和 `from`：

```
≃-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}
    }
```

如上，`to` 和 `from` 都是恒等函数，`from∘to` 和 `to∘from` 都是丢弃参数、返回 `refl` 的函数。在这样的情况下，`refl` 足够可以证明左逆，因为 `from (to x)` 化简为 `x`。右逆的证明同理。

要证明同构是对称的，我们把 `to` 和 `from`、`from∘to` 和 `to∘from` 互换：

```
≃-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
    }
```

要证明同构是传递的，我们将 `to` 和 `from` 函数进行组合，并使用相等性论证来结合左逆和右逆：

```
≃-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
      ∎}
    }
```

同构的相等性论证

我们可以直接的构造一种同构的相等性论证方法。我们对之前的相等性论证定义进行修改。我们省略 `_≡⟨⟩_` 的定义，因为简单的同构比简单的相等性出现的少很多：

```
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
```

嵌入（Embedding）

我们同时也需要嵌入的概念，它是同构的弱化概念。同构要求证明两个类型之间的一一对应，而嵌入只需要第一种类型涵盖在第二种类型内，所以两个类型之间有一对多的对应关系。

嵌入的正式定义如下：

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

除了它缺少了 `to∘from` 字段以外，嵌入的定义和同构是一样的。因此，我们可以得知 `from` 是 `to` 的左逆，但是 `from` 不是 `to` 的右逆。

嵌入是自反和传递的，但不是对称的。证明与同构类似，不过去除了不需要的部分：

```
≲-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
       ∎}
     }
```

显而易见的是，如果两个类型相互嵌入，且其嵌入函数相互对应，那么它们是同构的。这个一种反对称性的弱化形式：

```
≲-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
      ∎}
  }
```

前三部分可以直接从嵌入中得来，最后一部分我们可以把 `B ≲ A` 中的左逆和 两个嵌入中的 `to` 与 `from` 部分的相等性来获得同构中的右逆。

嵌入的相等性论证

和同构类似，我们亦支持嵌入的相等性论证：

```
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
```

练习 `≃-implies-≲`（实践）

证明每个同构蕴涵了一个嵌入。

```
postulate
  ≃-implies-≲ : ∀ {A B : Set}
    → A ≃ B
      -----
    → A ≲ B
```

```
-- 请将代码写在此处。
```

练习 `_⇔_`（实践）

按下列形式定义命题的等价性（又名“当且仅当“）：

```
record _⇔_ (A B : Set) : Set where
  field
    to : A → B
    from : B → A
```

证明等价性是自反、对称和传递的。

```
-- 请将代码写在此处。
```

练习 `Bin-embedding`（延伸）

回忆练习 Bin 和 Bin-laws，我们定义了比特串数据类型 `Bin` 来表示自然数，并要求你来定义下列函数：

```
to : ℕ → Bin
from : Bin → ℕ
```

它们满足如下性质：

```
from (to n) ≡ n
```

使用上述条件，证明存在一个从 `ℕ` 到 `Bin` 的嵌入。

```
-- 请将代码写在此处。
```

为什么 `to` 和 `from` 不能构造一个同构？

标准库

标准库中可以找到与本章节中相似的定义：

```
import Function using (_∘_)
import Function.Inverse using (_↔_)
import Function.LeftInverse using (_↞_)
```

标准库中的 `_↔_` 和 `_↞_` 分别对应了我们定义的 `_≃_` 和 `_≲_`，但是标准库中的定义使用起来不如我们的定义方便，因为标准库中的定义依赖于一个嵌套的记录结构，并可以由任何相等性的记法来参数化。

Unicode

本章节使用了如下 Unicode：

```
∘  U+2218  环运算符 (\o, \circ, \comp)
λ  U+03BB  小写希腊字母 LAMBDA (\lambda, \Gl)
≃  U+2243  渐进相等 (\~-)
≲  U+2272  小于或等价于 (\<~)
⇔  U+21D4  左右双箭头 (\<=>)
```

Chapter 6

Connectives: 合取、析取与蕴涵

```
module plfa.part1.Connectives where
```

本章节介绍基础的逻辑运算符。我们使用逻辑运算符与数据类型之间的对应关系，即命题即类型原理（Propositions as Types）。

- 合取（Conjunction）即是积（Product）
- 析取（Disjunction）即是和（Sum）
- 真（True）即是单元类型（Unit Type）
- 假（False）即是空类型（Empty Type）
- 蕴涵（Implication）即是函数空间（Function Space）

导入

```
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
```

合取即是积

给定两个命题 `A` 和 `B`，其合取 `A × B` 成立当 `A` 成立和 `B` 成立。我们将这样的概念形式化，使用如下的记录类型：

```
data _×_ (A B : Set) : Set where

  ⟨_,_⟩ :
      A
    → B
      -----
    → A × B
```

`A × B` 成立的证明由 `⟨ M , N ⟩` 的形式表现，其中 `M` 是 `A` 成立的证明， `N` 是 `B` 成立的证明。

给定 `A × B` 成立的证明，我们可以得出 `A` 成立和 `B` 成立。

```
proj₁ : ∀ {A B : Set}
  → A × B
    -----
  → A
proj₁ ⟨ x , y ⟩ = x

proj₂ : ∀ {A B : Set}
  → A × B
    -----
  → B
proj₂ ⟨ x , y ⟩ = y
```

如果 `L` 是 `A × B` 成立的证据，那么 `proj₁ L` 是 `A` 成立的证据， `proj₂ L` 是 `B` 成立的证据。

当 `⟨_,_⟩` 在等式右手边的项中出现的时候，我们将其称作*构造子*（Constructor），当它出现在等式左边时，我们将其称作*析构器*（Destructor）。我们亦可将 `proj₁` 和 `proj₂` 称作析构器，因为它们起到相似的效果。

其他的术语将 `⟨_,_⟩` 称作*引入*（Introduce）合取，将 `proj₁` 和 `proj₂` 称作*消去*（Eliminate）合取。前者亦记作 `×-I`，后者 `×-E₁` 和 `×-E₂`。如果我们从上到下来阅读这些规则，引入和消去正如其名字所说的那样：第一条*引入*一个运算符，所以运算符出现在结论中，而不是假设中；第二条*消去*一个带有运算符的式子，而运算符出现在假设中，而不是结论中。引入规则描述了运算符在什么情况下成立——即怎么样*定义*一个运算符。消去规则描述了运算符成立时，可以得出什么样的结论——即怎么样*使用*一个运算符。[1]

在这样的情况下，先使用析构器，再使用构造子将结果重组，得到还是原来的积。

```
η-× : ∀ {A B : Set} (w : A × B) → ⟨ proj₁ w , proj₂ w ⟩ ≡ w
η-× ⟨ x , y ⟩ = refl
```

左手边的模式匹配是必要的。用 `⟨ x , y ⟩` 来替换 `w` 让等式的两边可以化简成相同的项。

我们设置合取的优先级，使它与除了析取之外结合的都不紧密：

[1] 此段内容由 Propositions as Types（命题即类型）改编而来， 作者：Philip Wadler，发表于 《ACM 通讯》，2015 年 9 月

```
infixr 2 _×_
```

因此，`m ≤ n × n ≤ p` 解析为 `(m ≤ n) × (n ≤ p)`。

Alternatively, we can declare conjunction as a record type:

```
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 `_×_` 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.

给定两个类型 `A` 和 `B`，我们将 `A × B` 称为`A` 与 `B` 的积。在集合论中它也被称作笛卡尔积（Cartesian Product），在计算机科学中它对应记录类型。 如果类型 `A` 有`m` 个不同的成员，类型 `B` 有 `n` 个不同的成员，那么类型 `A × B` 有 `m * n` 个不同的成员。这也是它被称为积的原因之一。 例如，考虑有两个成员的`Bool` 类型，和有三个成员的 `Tri` 类型：

```
data Bool : Set where
  true : Bool
  false : Bool

data Tri : Set where
  aa : Tri
  bb : Tri
  cc : Tri
```

那么，`Bool × Tri` 类型有如下的六个成员：

```
⟨ true  , aa ⟩   ⟨ true  , bb ⟩   ⟨ true ,  cc ⟩
⟨ false , aa ⟩   ⟨ false , bb ⟩   ⟨ false , cc ⟩
```

下面的函数枚举了所有类型为 `Bool × Tri` 的参数：

```
×-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
```

类型上的积与数的积有相似的性质——它们满足交换律和结合律。更确切地说，积在在同构意义下满足交换律和结合率。

对于交换律，`to` 函数将有序对交换，将 `⟨ x , y ⟩` 变为 `⟨ y , x ⟩`，`from` 函数亦是如此（忽略命名）。在 `from∘to` 和 `to∘from` 中正确地实例化要匹配的模式是很重要的。 使用 `λ w → refl` 作为 `from∘to` 的定义是不可行的， `to∘from` 同理。

```
×-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 }
    }
```

满足交换律和在同构意义下满足交换律是不一样的。比较下列两个命题：

```
m * n ≡ n * m
A × B ≃ B × A
```

在第一个情况下，我们可能有 `m` 是 `2`、`n` 是 `3`，那么 `m * n` 和 `n * m` 都是 `6`。在第二个情况下，我们可能有 `A` 是 `Bool` 和 `B` 是 `Tri`，但是 `Bool × Tri` 和 `Tri × Bool` 不是一样的。但是存在一个两者之间的同构。例如：`⟨ true , aa ⟩` 是前者的成员， 其对应后者的成员 `⟨ aa , true ⟩`。

对于结合律来说，`to` 函数将两个有序对进行重组：将 `⟨ ⟨ x , y ⟩ , z ⟩` 转换为 `⟨ x , ⟨ y , z ⟩ ⟩`，`from` 函数则为其逆。同样，左逆和右逆的证明需要在一个合适的模式来匹配，从而可以直接化简：

```
×-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 }
    }
```

满足结合律和在同构意义下满足结合律是不一样的。比较下列两个命题：

```
(m * n) * p ≡ m * (n * p)
(A × B) × C ≃ A × (B × C)
```

举个例子，`(ℕ × Bool) × Tri` 与 `ℕ × (Bool × Tri)` 不同，但是两个类型之间 存在同构。例如 `⟨ ⟨ 1 , true ⟩ , aa ⟩`，一个前者的成员，与`⟨ 1 , ⟨ true , aa ⟩ ⟩`， 一个后者的成员，相对应。

练习 `⇔≃×` （推荐）

证明之前定义的 `A ⇔ B` 与`(A → B) × (B → A)` 同构。

```
-- 请将代码写在此处。
```

真即是单元类型

恒真 `⊤` 恒成立。我们将这个概念用合适的记录类型来形式化：

```
data ⊤ : Set where

  tt :
    --
    ⊤
```

`⊤` 成立的证明由 `tt` 的形式构成。

恒真有引入规则，但没有消去规则。给定一个 `⊤` 成立的证明，我们不能得出任何有趣的结论。因为恒真恒成立，知道恒真成立不会给我们带来新的知识。

`η-×` 的 零元形式是 `η-⊤`，其断言了任何 `⊤` 类型的值一定等于 `tt`：

```
η-⊤ : ∀ (w : ⊤) → tt ≡ w
η-⊤ tt = refl
```

左手边的模式匹配是必要的。将 `w` 替换为 `tt` 让等式两边可以化简为相同的值。

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′ = _
```

我们将 `⊤` 称为单元类型（Unit Type）。实际上，`⊤` 类型只有一个成员 `tt`。例如，下面的函数枚举了所有 `⊤` 类型的参数：

->

```
⊤-count : ⊤ → ℕ
⊤-count tt = 1
```

对于数来说，`1` 是乘法的幺元。对应地，单元是积的幺元（在同构意义下）。对于左幺元来说，`to` 函数将 `⟨ tt , x ⟩` 转换成 `x`，`from` 函数则是其反函数。左逆的证明需要匹配一个合适的模式来化简：

```
⊤-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 }
    }
```

幺元和在同构意义下的幺元是不一样的。比较下列两个命题：

```
1 * m ≡ m
⊤ × A ≃ A
```

在第一种情况下，我们可能有 `m` 是 `2`，那么 `1 * m` 和 `m` 都为 `2`。在第二种情况下，我们可能有 `A` 是 `Bool`，但是 `⊤ × Bool` 和 `Bool` 是不同的。例如：`⟨ tt , true ⟩` 是前者的成员，其对应后者的成员 `true`。

右幺元可以由积的交换律得来：

```
⊤-identityʳ : ∀ {A : Set} → (A × ⊤) ≃ A
⊤-identityʳ {A} =
  ≃-begin
    (A × ⊤)
  ≃⟨ ×-comm ⟩
    (⊤ × A)
  ≃⟨ ⊤-identityˡ ⟩
    A
  ≃-∎
```

我们在此使用了同构链，与等式链相似。

析取即是和

给定两个命题 `A` 和 `B`，析取 `A ⊎ B` 在 `A` 成立或者 `B` 成立时成立。我们将这个概念用合适的归纳类型来形式化：

```
data _⊎_ (A B : Set) : Set where

  inj₁ :
    A
    -----
    → A ⊎ B

  inj₂ :
    B
    -----
    → A ⊎ B
```

`A ⊎ B` 成立的证明有两个形式：`inj₁ M`，其中 `M` 是 `A` 成立的证明，或者 `inj₂ N`，其中 `N` 是 `B` 成立的证明。

给定 `A → C` 和 `B → C` 成立的证明，那么给定一个 `A ⊎ B` 的证明，我们可以得出 `C` 成立：

```
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
```

对 `inj₁` 和 `inj₂` 进行模式匹配，是我们使用析取成立的证明的常见方法。

当 `inj₁` 和 `inj₂` 在等式右手边出现的时候，我们将其称作构造子，当它出现在等式左边时，我们将其称作析构器。我们亦可将 `case-⊎` 称作析构器，因为它们起到相似的效果。其他术语将 `inj₁` 和 `inj₂` 称为引入析取， 将 `case-⊎` 称为消去析取。前者亦被称为 `⊎-I₁` 和 `⊎-I₂`，后者 `⊎-E`。

对每个构造子使用析构器得到的是原来的值：

```
η-⊎ : ∀ {A B : Set} (w : A ⊎ B) → case-⊎ inj₁ inj₂ w ≡ w
η-⊎ (inj₁ x) = refl
η-⊎ (inj₂ y) = refl
```

更普遍地来说，我们亦可对于析取使用一个任意的函数：

```
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
```

左手边的模式匹配是必要的。用 `inj₁ x` 来替换 `w` 让等式的两边可以化简成相同的项， `inj₂ y` 同理。

我们设置析取的优先级，使它与任何已经定义的运算符都结合的不紧密：

```
infixr 1 _⊎_
```

因此 `A × C ⊎ B × C` 解析为 `(A × C) ⊎ (B × C)`。

给定两个类型 `A` 和 `B`，我们将 `A ⊎ B` 称为 `A` 与 `B` 的和。在集合论中它也被称作不交并（Disjoint Union），在计算机科学中它对应变体记录类型。 如果类型 `A` 有 `m` 个不同的成员，类型 `B` 有 `n` 个不同的成员，那么类型 `A ⊎ B` 有 `m + n` 个不同的成员。这也是它被称为和的原因之一。 例如，考虑有两个成员的 `Bool` 类型，和有三个成员的 `Tri` 类型，如之前的定义。那么， `Bool ⊎ Tri` 类型有如下的五个成员：

```
inj₁ true     inj₂ aa
inj₁ false    inj₂ bb
              inj₂ cc
```

下面的函数枚举了所有类型为 `Bool ⊎ Tri` 的参数：

```
⊎-count : Bool ⊎ Tri → ℕ
⊎-count (inj₁ true)  = 1
⊎-count (inj₁ false) = 2
⊎-count (inj₂ aa)    = 3
⊎-count (inj₂ bb)    = 4
⊎-count (inj₂ cc)    = 5
```

类型上的和与数的和有相似的性质——它们满足交换律和结合律。更确切地说，和在在同构意义下是交换和结合的。

练习 `⊎-comm` （推荐）

证明和类型在同构意义下满足交换律。

```
-- 请将代码写在此处。
```

练习 `⊎-assoc` （实践）

证明和类型在同构意义下满足结合律。

```
-- 请将代码写在此处。
```

假即是空类型

恒假 `⊥` 从不成立。我们将这个概念用合适的归纳类型来形式化：

```
data ⊥ : Set where
  -- 没有语句!
```

没有 `⊥` 成立的证明。

与 `⊤` 相对偶，`⊥` 没有引入规则，但是有消去规则。因为恒假从不成立，如果它一旦成立，我们就进入了矛盾之中。给定 `⊥` 成立的证明，我们可以得出任何结论！这是逻辑学的基本原理，又由中世纪的拉丁文词组 *ex falso* 为名。小孩子也由诸如“如果猪有翅膀，那我就是示巴女王”的词组中知晓。我们如下将它形式化：

```
⊥-elim : ∀ {A : Set}
  → ⊥
    --
  → A
⊥-elim ()
```

这是我们第一次使用荒谬模式（Absurd Pattern）`()`。在这里，因为 `⊥` 是一个没有成员的类型，我们用 `()` 模式来指明这里不可能匹配任何这个类型的值。

`case-⊎` 的零元形式是 `⊥-elim`。类比的来说，它应该叫做 `case-⊥`，但是我们在此使用标准库中使用的名字。

`uniq-⊎` 的零元形式是 `uniq-⊥`，其断言了 `⊥-elim` 和任何取 `⊥` 的函数是等价的。

```
uniq-⊥ : ∀ {C : Set} (h : ⊥ → C) (w : ⊥) → ⊥-elim w ≡ h w
uniq-⊥ h ()
```

使用荒谬模式断言了 `w` 没有任何可能的值，因此等式显然成立。

```
⊥-count : ⊥ → ℕ
⊥-count ()
```

同样，荒谬模式告诉我们没有值可以来匹配类型 `⊥`。

对于数来说，0 是加法的幺元。对应地，空是和的幺元（在同构意义下）。

练习 `⊥-identityˡ`（推荐）

证明空在同构意义下是和的左幺元。

```
-- 请将代码写在此处。
```

Exercise `⊥-identityʳ` (practice)

练习 `⊥-identityʳ`（实践）

证明空在同构意义下是和的右幺元。

```
-- 请将代码写在此处。
```

蕴涵即是函数

给定两个命题 `A` 和 `B`，其蕴涵 `A → B` 在任何 `A` 成立的时候，`B` 也成立时成立。我们用函数类型来形式化蕴涵，如本书中通篇出现的那样。

`A → B` 成立的证据由下面的形式组成：

```
λ (x ꞉ A) → N
```

其中 `N` 是一个类型为 `B` 的项，其包括了一个类型为 `A` 的自由变量 `x`。给定一个 `A → B` 成立的证明 `L`，和一个 `A` 成立的证明 `M`，那么 `L M` 是 `B` 成立的证明。也就是说，`A → B` 成立的证明是一个函数，将 `A` 成立的证明转换成 `B` 成立的证明。

换句话说，如果知道 `A → B` 和 `A` 同时成立，那么我们可以推出 `B` 成立：

```
→-elim ꞉ ∀ {A B ꞉ Set}
  → (A → B)
  → A
    -------
  → B
→-elim L M = L M
```

在中世纪，这条规则被叫做 *modus ponens*，它与函数应用相对应。

定义一个函数，不管是带名字的定义或是使用 Lambda 抽象，被称为引入一个函数，使用一个函数被称为消去一个函数。

引入后接着消去，得到的还是原来的值：

```
η-→ ꞉ ∀ {A B ꞉ Set} (f ꞉ A → B) → (λ (x ꞉ A) → f x) ≡ f
η-→ f = refl
```

蕴涵比其他的运算符结合得都不紧密。因此 `A ⊎ B → B ⊎ A` 被解析为 `(A ⊎ B) → (B ⊎ A)`。

给定两个类型 `A` 和 `B`，我们将 `A → B` 称为从 `A` 到 `B` 的函数空间。它有时也被称作以 `B` 为底，`A` 为次数的幂。如果类型 `A` 有 `m` 个不同的成员，类型 `B` 有 `n` 个不同的成员，那么类型 `A → B` 有 n^m 个不同的成员。这也是它被称为幂的原因之一。例如，考虑有两个成员的 `Bool` 类型，和有三个成员的 `Tri` 类型，如之前的定义。那么，`Bool → Tri` 类型有如下的九个成员（三的平方）：

```
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}
```

下面的函数枚举了所有类型为 `Bool → Tri` 的参数：

```
→-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
```

类型上的幂与数的幂有相似的性质，很多数上成立的关系式也可以在类型上成立。

对应如下的定律：

```
(p ^ n) ^ m  ≡  p ^ (n * m)
```

我们有如下的同构：

```
A → (B → C)  ≃  (A × B) → C
```

两个类型可以被看作给定 `A` 成立的证据和 `B` 成立的证据，返回 `C` 成立的证据。这个同构有时也被称作柯里化（Currying）。右逆的证明需要外延性：

```
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 }}
    }
```

柯里化告诉我们，如果一个函数有取一个数据对作为参数，那么我们可以构造一个函数，取第一个参数，返回一个取第二个参数，返回最终结果的函数。因此，举例来说，下面表示加法的形式：

```
_+_ : ℕ → ℕ → ℕ
```

和下面的一个带有一个数据对作为参数的函数是同构的：

```
_+′_ : (ℕ × ℕ) → ℕ
```

Agda 对柯里化进行了优化，因此 `2 + 3` 是 `_+_ 2 3` 的简写。在一个对有序对进行优化的语言里，`2 +′ 3` 可能是 `_+′_ ⟨ 2 , 3 ⟩` 的简写。

对应如下的定律：

```
p ^ (n + m) = (p ^ n) * (p ^ m)
```

我们有如下的同构：

```
(A ⊎ B) → C  ≃  (A → C) × (B → C)
```

命题如果 `A` 成立或者 `B` 成立，那么 `C` 成立，和命题如果 `A` 成立，那么 `C` 成立以及 如果`B` 成立，那么 `C` 成立，是一样的。左逆的证明需要外延性：

```
→-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 }
    }
```

对应如下的定律：

```
(p * n) ^ m = (p ^ m) * (n ^ m)
```

我们有如下的同构：

```
A → B × C  ≃  (A → B) × (A → C)
```

命题如果 `A` 成立，那么 `B` 成立和 `C` 成立，和命题如果 `A` 成立，那么 `B` 成立以及 如果`A` 成立，那么 `C` 成立，是一样的。左逆的证明需要外延性和积的 `η-×` 规则：

```
→-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 }
    }
```

分配律

在同构意义下，积对于和满足分配律。验证这条形式的代码和之前的证明相似：

```
×-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
              }
    }
```

和对于积不满足分配律，但满足嵌入：

```
⊎-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
              }
    }
```

我们在定义 `from` 函数的时候可以有选择。给定的定义中，它将 `⟨ inj₂ z , inj₂ z′ ⟩` 转换为 `inj₂ z`，但我们也可以返回 `inj₂ z′` 作为嵌入证明的变种。我们在这里只能证明嵌入， 而不能证明同构，因为 `from` 函数必须丢弃 `z` 或者 `z′` 其中的一个。

在一般的逻辑学方法中，两条分配律都以等价的形式给出，每一边都蕴涵了另一边：

```
A × (B ⊎ C) ⇔ (A × B) ⊎ (A × C)
A ⊎ (B × C) ⇔ (A ⊎ B) × (A ⊎ C)
```

但当我们考虑提供上述蕴涵证明的函数时，第一条对应同构而第二条只能对应嵌入，揭示了有些定理比另一个更加的”正确“。

练习 `⊎-weak-×` （推荐）

证明如下性质成立：

```
postulate
  ⊎-weak-× : ∀ {A B C : Set} → (A ⊎ B) × C → A ⊎ (B × C)
```

这被称为*弱分配律*。给出相对应的分配律，并解释分配律与弱分配律的关系。

```
-- 请将代码写在此处。
```

练习 `⊎×-implies-×⊎` （实践）

证明合取的析取蕴涵了析取的合取：

```
postulate
  ⊎×-implies-×⊎ : ∀ {A B C D : Set} → (A × B) ⊎ (C × D) → (A ⊎ C) × (B ⊎ D)
```

反命题成立吗？如果成立，给出证明；如果不成立，给出反例。

```
-- 请将代码写在此处。
```

标准库

标准库中可以找到与本章节中相似的定义：

```
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 (_⇔_)
```

标准库中使用 `_,_` 构造数据对，而我们使用`⟨_,_⟩`。前者在从数据对构造三元对或者更大的元组时更加的方便，允许 `a , b , c` 作为`(a, (b , c))` 的记法。但它与其他有用的记法相冲突，比如说 Lists 中的 `[_,_]` 记法表示两个元素的列表， 或者 DeBruijn 章节中的 `Γ , A` 来表示环境的扩展。 标准库中的 `_⇔_` 和我们的相似，但使用起来比较不便，因为它可以根据任意的相等性定义进行参数化。

Unicode

本章节使用下列 Unicode：

```
×  U+00D7  乘法符号 (\x)
⊎  U+228E  多重集并集 (\u+)
⊤  U+22A4  向下图钉 (\top)
⊥  U+22A5  向上图钉 (\bot)
η  U+03B7  希腊小写字母 ETA (\eta)
₁  U+2081  下标 1 (\_1)
₂  U+2082  下标 2 (\_2)
⇔  U+21D4  左右双箭头 (\<=>)
```

Chapter 7

Negation: 直觉逻辑与命题逻辑中的否定

```
module plfa.part1.Negation where
```

本章介绍了否定的性质，讨论了直觉逻辑和经典逻辑。

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)
```

否定

给定命题 `A`，当 `A` 不成立时，它的否定形式 `¬ A` 成立。 我们将否定阐述为「蕴涵假」来形式化此概念。

```
¬_ : Set → Set
¬ A = A → ⊥
```

这是**归谬法（Reductio ad Absurdum）**的一种形式：若从 `A` 可得出结论 `⊥`（即喵呜），则 `¬ A` 必定成立。

`¬ A` 成立的证据的形式为：

```
λ{ x → N }
```

其中 `N` 是类型为 `⊥` 的项，它包含类型为 `A` 的自由变量 `x`。换言之，`¬ A` 成立的证据是一个函数，该函数将 `A` 成立的证据转换为 `⊥` 成立的证据。

给定 `¬ A` 和 `A` 均成立的证据，我们可以得出 `⊥` 成立。换言之，若 `¬ A` 和 `A` 均成立，那么我们就得到了矛盾：

```
¬-elim : ∀ {A : Set}
  → ¬ A
  → A
    ---
  → ⊥
¬-elim ¬x x = ¬x x
```

在这里，我们将 `¬ A` 的证据写作 `¬x`，将 `A` 的证据写作 `x`。这表示 `¬x` 必须是类型为 `A → ⊥` 的函数，因此应用 `¬x x` 得到的类型必为 `⊥`。注意此规则只是 `→-elim` 的一个特例。

我们将否定的优先级设定为高于析取和合取，但低于其它运算：

```
infix 3 ¬_
```

因此，`¬ A × ¬ B` 会解析为 `(¬ A) × (¬ B)`，而 `¬ m ≡ n` 会解析为 `¬ (m ≡ n)`。

在**经典逻辑**中，`A` 等价于 `¬ ¬ A`。而如前文所述，Agda 中使用了**直觉逻辑**，因此我们只有该等价关系的一半，即 `A` 蕴涵 `¬ ¬ A`：

```
¬¬-intro : ∀ {A : Set}
  → A
    -----
  → ¬ ¬ A
¬¬-intro x = λ{¬x → ¬x x}
```

令 `x` 为 `A` 的证据。我们要证明若假定 `¬ A` 成立，则会导出矛盾，因此 `¬ ¬ A` 必定成立。令 `¬x` 为 `¬ A` 的证据。那么以 `¬x x` 为证据，从 `A` 和 `¬ A` 可以导出矛盾。 这样我们就证明了 `¬ ¬ A`。

以上描述的等价写法如下：

```
¬¬-intro′ : ∀ {A : Set}
  → A
    -----
  → ¬ ¬ A
¬¬-intro′ x ¬x = ¬x x
```

在这里我们简单地将 λ-项的参数转换成了该函数的额外参数。我们通常会使用后面这种形式，因为它更加紧凑。

我们无法证明 `¬ ¬ A` 蕴涵 `A`，但可以证明 `¬ ¬ ¬ A` 蕴涵 `¬ A`：

```
¬¬¬-elim : ∀ {A : Set}
  → ¬ ¬ ¬ A
    -------
  → ¬ A
¬¬¬-elim ¬¬¬x = λ x → ¬¬¬x (¬¬-intro x)
```

令 `¬¬¬x` 为 `¬ ¬ ¬ A` 的证据。我们要证明若假定 `A` 成立就会导出矛盾，因此 `¬ A` 必定成立。令 `x` 为 `A` 的证据。根据前面的结果，以 `¬¬-intro x` 为证据可得出结论 `¬ ¬ A`。根据 `¬¬¬x (¬¬-intro x)`，我们可从 `¬ ¬ ¬ A` 和 `¬ ¬ A` 导出矛盾。这样我们就证明了 `¬ A`。

另一个逻辑规则是**换质换位律（contraposition）**，它陈述了若 `A` 蕴涵 `B`，则 `¬ B` 蕴涵 `¬ A`：

```
contraposition : ∀ {A B : Set}
  → (A → B)
    -----------
  → (¬ B → ¬ A)
contraposition f ¬y x = ¬y (f x)
```

令 `f` 为 `A → B` 的证据，`¬y` 为 `¬ B` 的证据。我们要证明，若假定 `A` 成立就会导出矛盾，因此 `¬ A` 必定成立。令 `x` 为 `A` 的证据。根据 `f x`，我们可从 `A → B` 和 `A` 我们可得出结论 `B`。而根据 `¬y (f x)`，可从 `B` 和 `¬ B` 得出结论 `⊥`。这样，我们就证明了 `¬ A`。

利用否定可直接定义不等性：

```
_≠_ : ∀ {A : Set} → A → A → Set
x ≠ y = ¬ (x ≡ y)
```

要证明不同的数不相等很简单：

```
_ : 1 ≠ 2
_ = λ()
```

这是我们第一次在 λ-表达式中使用谬模式（Absurd Pattern）。类型 `M ≡ N` 只有在 `M` 和 `N` 可被化简为相同的项时才能居留。由于 `1` 和 `2` 会化简为不同的正规形式，因此 Agda 判定没有证据可证明 `1 ≡ 2`。第二个例子是，很容易验证皮亚诺公理中「零不是任何数的后继数」的假设：

```
peano : ∀ {m : ℕ} → zero ≠ suc m
peano = λ()
```

它们的证明基本上相同，因为谬模式会匹配所有类型为 `zero ≡ suc m` 的可能的证据。

鉴于蕴涵和幂运算之间的对应关系，以及没有成员的类型为假，我们可以将否定看作零的幂。它确实对应于我们所知的算术运算，即

```
0 ^ n  ≡  1,  if n ≡ 0
       ≡  0,  if n ≢ 0
```

确实，只有一个 `⊥ → ⊥` 的证明。我们可以用两种方式写出此证明：

```
id : ⊥ → ⊥
id x = x

id′ : ⊥ → ⊥
id′ ()
```

不过使用外延性，我们可以证明二者相等：

```
id≡id′ : id ≡ id′
id≡id′ = extensionality (λ())
```

根据外延性，对于任何在二者定义域中的 `x`，都有 `id x ≡ id′ x`，则 `id ≡ id′` 成立。不过没有 `x` 在它们的定义域中，因此其相等性平凡成立。

实际上，我们可以证明任意两个否定的证明都是相等的：

```
assimilation : ∀ {A : Set} (¬x ¬x′ : ¬ A) → ¬x ≡ ¬x′
assimilation ¬x ¬x′ = extensionality (λ x → ⊥-elim (¬x x))
```

`¬ A` 的证据蕴涵任何 `A` 的证据都可直接得出矛盾。但由于外延性全称量化了使 `A` 成立的 `x`，因此任何这样的 `x` 都会直接导出矛盾，同样其相等性平凡成立。

练习 `<-irreflexive`（推荐）

利用否定证明严格不等性满足非自反性，即 `n < n` 对于任何 `n` 都不成立。

```
-- 请将代码写在此处
```

练习 `trichotomy`（实践）

请证明严格不等性满足三分律，即对于任何自然数 `m` 和 `n`，以下三条刚好只有一条成立：

- `m < n`

- `m ≡ n`
- `m > n`

「刚好只有一条」的意思是，三者中不仅有一条成立，而且当其中一条成立时，其它二者的否定也必定成立。

```
-- 请将代码写在此处
```

练习 `⊎-dual-×`（推荐）

请证明合取、析取和否定可通过以下版本的德摩根定律（De Morgan's Law）关联在一起。

```
¬ (A ⊎ B) ≃ (¬ A) × (¬ B)
```

此结果是我们之前证明的定理的简单推论。

```
-- 请将代码写在此处
```

以下命题也成立吗?

```
¬ (A × B) ≃ (¬ A) ⊎ (¬ B)
```

若成立，请证明；若不成立，你能给出一个比同构更弱的关系将两边关联起来吗?

直觉逻辑与经典逻辑

在 Gilbert 和 Sullivan 的电影《船夫》(*The Gondoliers*）中，Casilda 被告知她还是个婴儿时，就被许配给了巴塔维亚国王的继承人。但由于一场动乱，没人知道她被许配给了两位继承人 Marco 和 Giuseppe 中的哪一位。她惊慌地哀嚎道:「那么你的意思是说我嫁给了两位船夫中的一位，但却无法确定是谁?」对此的回答是:「虽然不知道是谁，但这件事却是毫无疑问的。」

逻辑学有很多变种，而**经典逻辑**和**直觉逻辑**之间有一个区别。直觉主义者关注于某些逻辑学家对无限性本质的假设，坚持真理的构造主义的概念。具体来说，它们坚持认为 `A ⊎ B` 的证明必须确定 `A` 或 `B` 中的**哪一个**成立， 因此它们会解决宣称 Casilda 嫁给了 Marco 或者 Giuseppe，直到其中一个被确定为 她的丈夫为止。或许 Gilbert 和 Sullivan 期待直觉主义，因为在故事的结局中， 继承人是第三个人 Luiz，他和 Casilda 已经顺利地相爱了。

直觉主义者也拒绝排中律(Law of the Excluded Middle)————该定律断言,对于所有的 `A`，`A ⊎ ¬ A` 必定成立————因为该定律没有给出 `A` 和 `¬ A` 中的哪一个成立。海廷（Heyting）形式化了希尔伯特（Hilbert）经典逻辑的一个变种，抓住了直觉主义中可证明性的概念。具体来说，排中律在希尔伯特逻辑中是可证明的，但在海廷逻辑中却不可证明。进一步来说，如果排中律作为一条公理添加到海廷逻辑中，那么它会等价于希尔伯特逻辑。柯尔莫哥洛夫（Kolmogorov）证明了两种逻辑紧密相关：他给出了双重否定翻译，即一个式子在经典逻辑中可证，当且仅当它的双重否定式在直觉逻辑中可证。

「命题即类型」最初是为直觉逻辑而制定的。这是一种完美的契合，因为在直觉主义的解释中，式子 `A ⊎ B` 刚好可以在给出 `A` 或 `B` 之一的证明时得证，因此对应于析取 的类型是一个不交和（Disjoint Sum）。

（以上内容部分取自 “Propositions as Types”, Philip Wadler, *Communications of the ACM*，2015 年 12 月。）

排中律是不可辩驳的

排中律可形式化如下：

```
postulate
  em : ∀ {A : Set} → A ⊎ ¬ A
```

如之前所言，排中律在直觉逻辑中并不成立。然而，我们可以证明它是**不可辩驳（Irrefutable）**的，即其否定的否定是可证明的（因而其否定式不可证明）：

```
em-irrefutable : ∀ {A : Set} → ¬ ¬ (A ⊎ ¬ A)
em-irrefutable = λ k → k (inj₂ (λ x → k (inj₁ x)))
```

解释此代码的最佳方式是交互式地推导它：

```
em-irrefutable k = ?
```

给定 `¬ (A ⊎ ¬ A)` 的证据 `k`，即一个函数，它接受一个类型为 `A ⊎ ¬ A` 的值，返回一个空类型的值，我们必须在 `?` 处填上一个返回空类型的项。得到空类型值 的唯一方式就是应用 `k` 本身，于是我们据此展开此洞：

```
em-irrefutable k = k ?
```

我们需要用类型为 `A ⊎ ¬ A` 的值填上这个新的洞。由于目前我们并没有类型为 `A` 的值，因此先处理第二个析取：

```
em-irrefutable k = k (inj₂ λ{ x → ? })
```

第二个析取接受 `¬ A` 的证据，即一个函数，它接受类型为 `A` 的值，返回空类型的值。 我们将 `x` 绑定到类型为 `A` 的值，现在我们需要在洞中填入空类型的值。同样，得到空类型的值的唯一方法就是将 `k` 应用到其自身，于是我们展开此洞：

```
em-irrefutable k = k (inj₂ λ{ x → k ? })
```

这次我们就有一个类型为 `A` 的值了，其名为 `x`，于是我们可以处理第一个析取：

```
em-irrefutable k = k (inj₂ λ{ x → k (inj₁ x) })
```

现在没有洞了！这样就完成了证明。

下面的故事说明了我们创建的项的行为。（向 Peter Selinger 道歉，他讲的是个关于国王，巫师和贤者之石的类似的故事。）

曾经有一个恶魔向一个男人提议：「要么 (a) 我给你 10 亿美元，要么 (b) 如果你付给我 10 亿美元，我可以实现你的任何一个愿望。当然，得是我决定提供 (a) 还是 (b)。」

男人很谨慎。他需要付出他的灵魂吗？ 恶魔说不用，他只要接受这个提议就行。

于是男人思索着，如果恶魔向他提供 (b)，那么他不太可能付得起这个愿望。不过倘若真是如此的话，能有什么坏处吗？

「我接受」，男人回答道，「我能得到 (a) 还是 (b)？」

恶魔顿了顿。「我提供 (b)。」

男人很失望，但并不惊讶。「果然是这样」，他想。但是这个提议折磨着他。想想他都能用这个愿望做些什么！多年以后，男人开始积累钱财。为了得到这笔钱，他有时会做坏事，而且他隐约意识到这一定是魔鬼所想到的。最后他攒够了 10 亿美元，恶魔再次出现了。

「这是 10 亿美元」，男人说着，交出一个手提箱。「实现我的愿望吧！」

恶魔接过了手提箱。然后他说道，「哦？我之前说的是 (b) 吗？抱歉，我说的是 (a)。 很高兴能给你 10 亿美元。」

于是恶魔将那个手提箱又还给了他。

（以上内容部分取自 “Call-by-Value is Dual to Call-by-Name”, Philip Wadler, *International Conference on Functional Programming*, 2003 年。）

练习 `Classical`（延伸）

考虑以下定律：

- 排中律：对于所有 `A`，`A ⊎ ¬ A`。
- 双重否定消去：对于所有的 `A`，`¬ ¬ A → A`。
- 皮尔士定律：对于所有的 `A` 和 `B`，`((A → B) → A) → A`。
- 蕴涵表示为析取：对于所有的 `A` 和 `B`，`(A → B) → ¬ A ⊎ B`。
- 德摩根定律：对于所有的 `A` 和 `B`，`¬ (¬ A × ¬ B) → A ⊎ B`。

请证明其中任意一条定律都蕴涵其它所有定律。

```
-- 请将代码写在此处
```

联系 `Stable`（延伸）

若双重否定消去对某个式子成立，我们就说它是**稳定**（**Stable**）的：

```
Stable : Set → Set
Stable A = ¬ ¬ A → A
```

请证明任何否定式都是稳定的，并且两个稳定式的合取也是稳定的。

```
-- 请将代码写在此处
```

标准库

本章中的类似定义可在标准库中找到：

```
import Relation.Nullary using (¬_)
import Relation.Nullary.Negation using (contraposition)
```

Unicode

本章使用了以下 Unicode：

```
¬  U+00AC  否定符号 (\neg)
≢  U+2262  不等价于 (\==n)
```

Chapter 8

Quantifiers: 全称量词与存在量词

```
module plfa.part1.Quantifiers where
```

本章节介绍全称量化（Universal Quantification）和存在量化（Existential Quantification）。

导入

```
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)
```

全称量化

我们用依赖函数类型（Dependent Function Type）来形式化全称量化，这样的形式在书中贯穿始终。例如，在归纳一章中，我们证明了加法满足结合律：

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

它断言对于所有的自然数 `m`、`n` 和 `p`，`(m + n) + p ≡ m + (n + p)` 成立。它是一个依赖函数，给出 `m`、`n` 和 `p` 的值，它会返回与该等式对应的证据。

通常给定一个 `A` 类型的变量 `x` 和一个带有 `x` 自由变量的命题 `B x`，全称量化 的命题`∀ (x : A) → B x` 当对于所有类型为 `A` 的项`M`，命题 `B M` 成立时成立。 在这里，`B M` 代表了将 `B x` 中自由出现的变量 `x`

替换成 `M` 后的命题。 变量 `x` 在 `B x` 中以自由变量的形式出现，但是在 `∀ (x ∶ A) → B x` 中是约束的。

`∀ (x ∶ A) → B x` 成立的证明由以下形式构成：

```
λ (x ∶ A) → N x
```

其中 `N x` 是一个 `B x` 类型的项， `N x` 和 `B x` 都包含了一个 `A` 类型的自由变量 `x`。给定一个项 `L`， 其提供 `∀ (x ∶ A) → B x` 成立的证明，和一个类型为 `A` 的项 `M`， `L M` 这一项则是 `B M` 成立的证明。换句话说， `∀ (x ∶ A) → B x` 成立的证明是一个函数， 将类型为 `A` 的项 `M` 转换成 `B M` 成立的证明。

再换句话说，如果我们知道 `∀ (x ∶ A) → B x` 成立，又知道 `M` 是一个类型为 `A` 的项， 那么我们可以推导出 `B M` 成立：

```
∀-elim ∶ ∀ {A ∶ Set} {B ∶ A → Set}
  → (L ∶ ∀ (x ∶ A) → B x)
  → (M ∶ A)
    ----------------
  → B M
∀-elim L M = L M
```

如 `→-elim` 那样，这条规则对应了函数应用。

函数是依赖函数的一种特殊形式，其值域不取决于定义域中的变量。当一个函数被视为蕴涵的证明时，它的参数和结果都是证明，而当一个依赖函数被视为全称量词的证明时，它的参数被视为数据类型中的一个元素，而结果是一个依赖于参数的命题的证明。因为在 Agda 中，一个数据类型中的一个值和一个命题的证明是无法区别的，这样的区别很大程度上取决于如何来诠释。

依赖函数类型也被叫做依赖积（Dependent Product），因为如果 `A` 是一个有限的数据类型，有值 `x₁ , ⋯ , xₙ`，如果每个类型 `B x₁ , ⋯ , B xₙ` 有 `m₁ , ⋯ , mₙ` 个不同的成员， 那么 `∀ (x ∶ A) → B x` 有 `m₁ * ⋯ * mₙ` 个成员。的确， `∀ (x ∶ A) → B x` 的记法有时 也被 `Π[ x ∈ A ] (B x)` 取代，其中 `Π` 代表积。然而，我们还是使用依赖函数这个名称，因为依赖积这个名称是有歧义的，我们后续会体会到歧义所在。

练习 `∀-distrib-×` （推荐）

证明全称量词对于合取满足分配律：

```
postulate
  ∀-distrib-× ∶ ∀ {A ∶ Set} {B C ∶ A → Set} →
    (∀ (x ∶ A) → B x × C x) ≃ (∀ (x ∶ A) → B x) × (∀ (x ∶ A) → C x)
```

将这个结果与 Connectives 章节中的（`→-distrib-×`）结果对比。

练习 `⊎∀-implies-∀⊎`（实践）

证明全称命题的析取蕴涵了析取的全称命题：

```
postulate
  ⊎∀-implies-∀⊎ : ∀ {A : Set} {B C : A → Set} →
    (∀ (x : A) → B x) ⊎ (∀ (x : A) → C x) → ∀ (x : A) → B x ⊎ C x
```

逆命题成立么？如果成立，给出证明。如果不成立，解释为什么。

练习 `∀-×`（实践）

参考下面的类型：

```
data Tri : Set where
  aa : Tri
  bb : Tri
  cc : Tri
```

令 `B` 作为由 `Tri` 索引的一个类型，也就是说`B : Tri → Set`。 证明 `∀ (x : Tri) → B x` 和 `B aa × B bb × B cc` 是同构的。提示：你需要引入一个可应用于依赖函数的外延性公设。

存在量化

给定一个 `A` 类型的变量 `x` 和一个带有 `x` 自由变量的命题 `B x`，存在量化 的命题`Σ[ x ∈ A ] B x` 当对于一些类型为 `A` 的项`M`，命题 `B M` 成立时成立。 在这里，`B M` 代表了将 `B x` 中自由出现的变量 `x` 替换成 `M` 以后的命题。 变量 `x` 在 `B x` 中以自由变量形式出现，但是在 `Σ[ x ∈ A ] B x` 中是约束的。

我们定义一个合适的归纳数据类型来形式化存在量化：

```
data Σ (A : Set) (B : A → Set) : Set where
  ⟨_,_⟩ : (x : A) → B x → Σ A B
```

我们为存在量词定义一个方便的语法：

```
Σ-syntax = Σ
infix 2 Σ-syntax
syntax Σ-syntax A (λ x → B) = Σ[ x ∈ A ] B
```

这是我们第一次使用语法声明，其表示左手边的项可以以右手边的语法来书写。这种特殊语法只有在标识符 `Σ-syntax` 被导入时可用。

`Σ[ x ∈ A ] B x` 成立的证明由 `⟨ M , N ⟩` 组成，其中 `M` 是类型为 `A` 的项， `N` 是 `B M` 成立的证明。

我们也可以用记录类型来等价地定义存在量化。

```
record Σ′ (A : Set) (B : A → Set) : Set where
  field
    proj₁′ : A
    proj₂′ : B proj₁′
```

这里的记录构造

```
record
  { proj₁′ = M
  ; proj₂′ = N
  }
```

对应了项

```
⟨ M , N ⟩
```

其中 `M` 是类型为 `A` 的项， `N` 是类型为 `B M` 的项。

积是依赖积的一种特殊形式，其第二分量不取决于第一分量中的变量。当一个积被视为合取的证明时，它的两个分量都是证明，而当一个依赖积被视为存在量词的证明时，它的第一分量被视为数据类型中的一个元素，而第二分量是一个依赖于第一分量的命题的证明。因为在 Agda 中，一个数据类型中的一个值一个命题的证明是无法区别的，这样的区别很大程度上取决于如何来诠释。

存在量化也被叫做依赖和 (Dependent Sum)，因为如果 `A` 是一个有限的数据类型，有值 `x₁ , ⋯ , xₙ`，如果每个类型 `B x₁ , ⋯ , B xₙ` 有 `m₁ , ⋯ , mₙ` 个不同的成员，那么 `Σ[ x ∈ A ] B x` 有 `m₁ + ⋯ + mₙ` 个成员，这也解释了选择使用这个记法的原因——— `Σ` 代表和。

存在量化有时也被叫做依赖积 (Dependent Product)，因为积是其中的一种特殊形式。但是，这样的叫法非常让人困扰，因为全程量化也被叫做依赖积，而存在量化已经有依赖和的叫法。

存在量词的普通记法是 `∃` （与全程量词的 `∀` 记法相类似)。我们使用 Agda 标准库中的惯例，使用一种隐式申明约束变量定义域的记法。

```
∃ : ∀ {A : Set} (B : A → Set) → Set
∃ {A} B = Σ A B

∃-syntax = ∃
syntax ∃-syntax (λ x → B) = ∃[ x ] B
```

这种特殊的语法只有在 `∃-syntax` 标识符被导入时可用。我们将倾向于使用这种语法，因为它更短，而且看上去更熟悉。

给定 `∀ x → B x → C` 成立的证明，其中 `C` 不包括自由变量 `x`，给定 `∃[ x ] B x` 成立的证明，我们可以

推导出 `C` 成立。

```
∃-elim : ∀ {A : Set} {B : A → Set} {C : Set}
  → (∀ x → B x → C)
  → ∃[ x ] B x
    --------------
  → C
∃-elim f ⟨ x , y ⟩ = f x y
```

换句话说，如果我们知道对于任何 `A` 类型的 `x`，`B x` 蕴涵了 `C`，还知道对于某个类型 `A` 的 `x`，`B x` 成立，那么我们可以推导出 `C` 成立。这是因为我们可以先将 `∀ x → B x → C` 的证明对于 `A` 类型的 `x` 和 `B x` 类型的 `y` 实例化，而这样的值恰好可以由 `∃[ x ] B x` 的证明来提供。

的确，逆命题也成立，两者合起来构成一个同构：

```
∀∃-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 }}
    }
```

这可以被看做是将柯里化推广的结果。的确，建立这两者同构的证明与之前我们讨论蕴涵时给出的证明是一样的。

练习 `∃-distrib-⊎` （推荐）

证明存在量词对于析取满足分配律：

```
postulate
  ∃-distrib-⊎ : ∀ {A : Set} {B C : A → Set} →
    ∃[ x ] (B x ⊎ C x) ≃ (∃[ x ] B x) ⊎ (∃[ x ] C x)
```

练习 `∃×-implies-×∃` （实践）

证明合取的存在命题蕴涵了存在命题的合取：

```
postulate
  ∃×-implies-×∃ : ∀ {A : Set} {B C : A → Set} →
```

```
∃[ x ] (B x × C x) → (∃[ x ] B x) × (∃[ x ] C x)
```

逆命题成立么？如果成立，给出证明。如果不成立，解释为什么。

练习 `∃-⊎`（实践）

沿用练习 `∀-×` 中的 `Tri` 和 `B` 。 证明 `∃[ x ] B x` 与 `B aa ⊎ B bb ⊎ B cc` 是同构的。

一个存在量化的例子

回忆我们在 Relations 章节中定义的 `even` 和 `odd`：

```
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)
```

如果一个数是 0 或者它是奇数的后继，那么这个数是偶数。如果一个数是偶数的后继，那么这个数是奇数。

我们接下来要证明，一个数是偶数当且仅当这个数是一个数的两倍，一个数是奇数当且仅当这个数是一个数的两倍多一。换句话说，我们要证明的是：

`even n` 当且仅当 `∃[ m ] (    m * 2 ≡ n)`

`odd  n` 当且仅当 `∃[ m ] (1 + m * 2 ≡ n)`

惯例来说，我们往往将常数因子写在前面、将和里的常数项写在后面。但是这里我们没有按照惯例，而是反了过来，因为这样可以让证明更简单：

这是向前方向的证明：

```
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 ⟩
```

我们定义两个相互递归的函数。给定 `n` 是奇数或者是偶数的证明，我们返回一个数 `m`，以及`m * 2 ≡ n` 或者 `1 + m * 2 ≡ n` 的证明。我们根据 `n` 是奇数 或者是偶数的证明进行归纳：

- 如果这个数是偶数，因为它是 0，那么我们返回数据对 0 ，以及 0 的两倍是 0 的证明。
- 如果这个数是偶数，因为它是比一个奇数多 1，那么我们可以使用归纳假设，来获得一个数 `m` 和 `1 + m * 2 ≡ n` 的证明。我们返回数据对 `suc m` 以及 `suc m * 2 ≡ suc n` 的证明—— 我们可以直接通过替换 `n` 来得到证明。
- 如果这个数是奇数，因为它是一个偶数的后继，那么我们可以使用归纳假设，来获得一个数`m` 和 `m * 2 ≡ n` 的证明。我们返回数据对`suc m` 以及 `1 + m * 2 ≡ suc n` 的证明—— 我们可以直接通过替换 `n` 来得到证明。

这样，我们就完成了正方向的证明。

接下来是反方向的证明：

```
∃-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 ⟩)
```

给定一个是另一个数两倍的数，我们需要证明这个数是偶数。给定一个是另一个数两倍多一的数，我们需要证明这个数是奇数。我们对于存在量化的证明进行归纳。在偶数的情况，我们也需要考虑两种一个数是另一个数两倍的情况。

- 在偶数是 `zero` 的情况中，我们需要证明 `zero * 2` 是偶数，由 `even-zero` 可得。
- 在偶数是 `suc n` 的情况中，我们需要证明 `suc m * 2` 是偶数。归纳假设告诉我们， `1 + m * 2` 是奇数，那么所求证的结果由 `even-suc` 可得。
- 在偶数的情况中，我们需要证明 `1 + m * 2` 是奇数。归纳假设告诉我们，`m * 2` 是偶数，那么所求证的结果由`odd-suc` 可得。

这样，我们就完成了向后方向的证明。

练习 `∃-even-odd`（实践）

如果我们用 `2 * m` 代替 `m * 2`，`2 * m + 1` 代替 `1 + m * 2`，上述证明会变得复杂多少呢？ 用这种方法来重写 `∃-even` 和 `∃-odd`。

```
-- 请将代码写在此处。
```

练习 `∃-|-≤`（实践）

证明当且仅当存在一个 `x` 使得 `x + y ≡ z` 成立时 `y ≤ z` 成立。

```
-- 请将代码写在此处。
```

存在量化、全称量化和否定

存在量化的否定与否定的全称量化是同构的。考虑到存在量化是析构的推广，全称量化是合构的推广，这样的结果与析构的否定与否定的合构是同构的结果相似。

```
¬∃≃∀¬ : ∀ {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 }
    }
```

在 `to` 的方向，给定了一个 `¬ ∃[ x ] B x` 类型的值 `¬∃xy`，需要证明给定一个 `x` 的值，可以推导出 `¬ B x`。换句话说，给定一个 `B x` 类型的值 `y`，我们可以推导出假。将 `x` 和 `y` 合并起来我们就得到了 `∃[ x ] B x` 类型的值 `⟨ x , y ⟩`，对其使用 `¬∃xy` 即可获得矛盾。

在 `from` 的方向，给定了一个 `∀ x → ¬ B x` 类型的值 `∀¬xy`，需要证明从一个类型为 `∃[ x ] B x` 类型的值 `⟨ x , y ⟩`，我们可以推导出假。将 `∀¬xy` 使用于 `x` 之上，可以得到类型为 `¬ B x` 的值，对其使用 `y` 即可获得矛盾。

两个逆的证明很直接，其中有一个方向需要外延性。

练习 `∃¬-implies-¬∀` （推荐）

证明否定的存在量化蕴涵了全称量化的否定：

```
postulate
  ∃¬-implies-¬∀ : ∀ {A : Set} {B : A → Set}
    → ∃[ x ] (¬ B x)
      --------------
    → ¬ (∀ x → B x)
```

逆命题成立吗？如果成立，给出证明。如果不成立，解释为什么。

练习 `Bin-isomorphism` （延伸）

回忆在练习 Bin、Bin-laws 和 Bin-predicates 中，我们定义了比特串数据类型 `Bin` 来表示自然数，并要求你定义了下列函数和谓词：

```
to   : ℕ → Bin
from : Bin → ℕ
Can  : Bin → Set
```

以及证明了下列性质：

```
from (to n) ≡ n

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

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

使用上述，证明 `ℕ` 与 `∃[ b ](Can b)` 之间存在同构。

我们建议证明以下引理，它描述了对于给定的二进制数 `b`，`One b` 只有一个证明，`Can b`，也是如此。

```
≡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′
```

```
-- 请将代码写在此处。
```

标准库

标准库中可以找到与本章中相似的定义：

```
import Data.Product using (Σ, _,_, ∃, Σ-syntax, ∃-syntax)
```

Unicode

本章节使用下列 Unicode：

```
Π  U+03A0  大写希腊字母 PI (\Pi)
Σ  U+03A3  大写希腊字母 SIGMA (\Sigma)
∃  U+2203  存在 (\ex, \exists)
```

Chapter 9

Decidable: 布尔值与判定过程

```
module plfa.part1.Decidable where
```

我们有两种不同的方式来表示关系：一是表示为由关系成立的证明（Evidence）所构成的数据类型；二是表示为一个计算（Compute）关系是否成立的函数。在本章中，我们将探讨这两种方式之间的关系。我们首先研究大家熟悉的布尔值（Boolean）记法，但是之后我们会发现，相较布尔值记法，使用一种新的可判定性（Decidable）记法将会是更好的选择。

导入

```
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 (_⇔_)
```

证据 vs 计算

回忆我们在 Relations 章节中将比较定义为一个归纳数据类型，其提供了一个数小于或等于另外一个数的证明：

```
infix 4 _≤_

data _≤_ : ℕ → ℕ → Set where

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

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

举例来说，我们提供 `2 ≤ 4` 成立的证明，也可以证明没有`4 ≤ 2` 成立的证明。

```
2≤4 : 2 ≤ 4
2≤4 = s≤s (s≤s z≤n)

¬4≤2 : ¬ (4 ≤ 2)
¬4≤2 (s≤s (s≤s ()))
```

`()` 的出现表明了没有 `2 ≤ 0` 成立的证明：`z≤n` 不能匹配（因为 `2` 不是 `zero`），`s≤s` 也不能匹配（因为 `0` 不能匹配 `suc n`）。

作为替代的定义，我们可以定义一个大家可能比较熟悉的布尔类型：

```
data Bool : Set where
  true : Bool
  false : Bool
```

给定了布尔类型，我们可以定义一个两个数的函数在比较关系成立时来计算出`true`， 否则计算出 `false`：

```
infix 4 _≤ᵇ_

_≤ᵇ_ : ℕ → ℕ → Bool
zero ≤ᵇ n    = true
suc m ≤ᵇ zero = false
suc m ≤ᵇ suc n = m ≤ᵇ n
```

定义中的第一条与最后一条与归纳数据类型中的两个构造子相对应。因为对于任意的`m`，不可能出现 `suc m ≤ zero` 的证明，我们使用中间一条定义来表示。 举个例子，我们可以计算`2 ≤ᵇ 4` 成立，也可以计算 `4 ≤ᵇ 2` 不成立：

```
_ : (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
  ∎
```

在第一种情况中，我们需要两步来将第一个参数降低到 0，再用一步来计算出真，这对应着我们需要 使用两次 `s≤s` 和一次 `z≤n` 来证明 `2 ≤ 4` 。在第二种情况中，我们需要两步来将第二个参数降低到 0，再用一步来计算出假，这对应着我们需要 使用两次 `s≤s` 和一次 `()` 来说明没有 `4 ≤ 2` 的证明。

将证明与计算相联系

我们希望能够证明这两种方法是有联系的，而我们的确可以。首先，我们定义一个函数来把计算世界映射到证明世界：

```
T : Bool → Set
T true  = ⊤
T false = ⊥
```

回忆到 `⊤` 是只有一个元素 `tt` 的单元类型，`⊥` 是没有值的空类型。（注意 `T` 是大写字母 `t`， 与 `⊤` 不同。）如果 `b` 是 `Bool` 类型的，那么如果 `b` 为真， `tt` 可以提供 `T b` 成立的证明； 如果 `b` 为假，则不可能有 `T b` 成立的证明。

换句话说， `T b` 当且仅当 `b ≡ true` 成立时成立。在向前的方向，我们需要针对 `b` 进行情况分析：

```
T→≡ : ∀ (b : Bool) → T b → b ≡ true
T→≡ true tt = refl
T→≡ false ()
```

如果 `b` 为真，那么 `T b` 由 `tt` 证明，`b ≡ true` 由 `refl` 证明。 当 `b` 为假，那么 `T b` 无法证明。

在向后的方向，不需要针对布尔值 `b` 的情况分析：

```
≡→T : ∀ {b : Bool} → b ≡ true → T b
≡→T refl = tt
```

如果 `b ≡ true` 由 `refl` 证明，我们知道 `b` 是 `true`，因此 `T b` 由 `tt` 证明。

现在我们可以证明 `T (m ≤ᵇ n)` 当且仅当 `m ≤ n` 成立时成立。

在向前的方向，我们考虑 `_≤ᵇ_` 定义中的三条语句：

```
≤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)
```

第一条语句中，我们立即可以得出 `zero ≤ᵇ n` 为真，所以`T (m ≤ᵇ n)` 由 `tt` 而得， 相对应地 `m ≤ n` 由 `z≤n` 而证明。在中间的语句中，我们立刻得出`suc m ≤ᵇ zero` 为假，则 `T (m ≤ᵇ n)` 为空，因此我们无需证明`m ≤ n`，同时也不存在这样的证明。在最后的语句中，我们对于`suc m ≤ᵇ suc n` 递归至 `m ≤ᵇ n`。令 `t` 为`T (suc m ≤ᵇ suc n)` 的证明，如果其存在。 根据`_≤ᵇ_` 的定义，这也是 `T (m ≤ᵇ n)` 的证明。我们递归地应用函数来获得 `m ≤ n` 的证明，再使用`s≤s` 将其转换成为 `suc m ≤ suc n` 的证明。

在向后的方向，我们考虑 `m ≤ n` 成立证明的可能形式：

```
≤→≤ᵇ : ∀ {m n : ℕ} → m ≤ n → T (m ≤ᵇ n)
≤→≤ᵇ z≤n       = tt
≤→≤ᵇ (s≤s m≤n) = ≤→≤ᵇ m≤n
```

如果证明是 `z≤n`，我们立即可以得到 `zero ≤ᵇ n` 为真，所以 `T (m ≤ᵇ n)` 由 `tt` 证明。 如果证明是`s≤s` 作用于 `m≤n`，那么 `suc m ≤ᵇ suc n` 规约到 `m ≤ᵇ n`，我们可以递归地使用函数 来获得`T (m ≤ᵇ n)` 的证明。

向前方向的证明比向后方向的证明多一条语句，因为在向前方向的证明中我们需要考虑比较结果为真和假的语句，而向后方向的证明只需要考虑比较成立的语句。这也是为什么我们比起计算的形式，更加偏爱证明的形式，因为这样让我们做更少的工作：我们只需要考虑关系成立时的情况，而可以忽略不成立的情况。

从另一个角度来说，有时计算的性质可能正是我们所需要的。面对一个大数值上的非显然关系，使用电脑来计算出答案可能会更加方便。幸运的是，比起在证明或计算之中犹豫，我们有一种更好的方法来兼取其优。

取二者之精华

一个返回布尔值的函数提供恰好一比特的信息：这个关系成立或是不成立。相反地，证明的形式告诉我们为什么这个关系成立，但却需要我们自行完成这个证明。不过，我们其实可以简单地定义一个类型来取二者之精华。我们把它叫做：`Dec A`，其中 `Dec` 是可判定的（Decidable）的意思。

```
data Dec (A : Set) : Set where
  yes : A → Dec A
  no : ¬ A → Dec A
```

正如布尔值，这个类型有两个构造子。一个 `Dec A` 类型的值要么是以 `yes x` 的形式，其中 `x` 提供 `A` 成立的证明，或者是以 `no ¬x` 的形式，其中 `x` 提供了 `A` 无法成立的证明。（也就是说，`¬x` 是一个给定 `A` 成立的证据，返回矛盾的函数）

比如说，我们定义一个函数 `_≤?_`，给定两个数，判定是否一个数小于等于另一个，并提供证明来说明结论。

首先，我们使用两个有用的函数，用于构造不等式不成立的证明：

```
¬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
```

第一个函数断言了 `¬ (suc m ≤ zero)`，由荒谬可得。因为每个不等式的成立证明必须是 `zero ≤ n` 或者 `suc m ≤ suc n` 的形式，两者都无法匹配 `suc m ≤ zero`。第二个函数取 `¬ (m ≤ n)` 的证明 `¬m≤n`，返回 `¬ (suc m ≤ suc n)` 的证明。所有形如 `suc m ≤ suc n` 的证明必须是以 `s≤s m≤n` 的形式给出。因此我们可以构造一个 矛盾，以 `¬m≤n m≤n` 来证明。

使用这些，我们可以直接的判定不等关系：

```
_≤?_ : ∀ (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)
```

与 `_≤ᵇ_` 一样，定义有三条语句。第一条语句中，`zero ≤ n` 立即成立，由 `z≤n` 证明。第二条语句中，`suc m ≤ zero` 立即不成立，由 `¬s≤z` 证明。第三条语句中，我们需要递归地应用 `m ≤? n`。有两种可能性，在 `yes` 的情况中，它会返回 `m ≤ n` 的证明 `m≤n`，所以 `s≤s m≤n` 即可作为 `suc m ≤ suc n` 的证明；在 `no` 的情况中，它会返回 `¬ (m ≤ n)` 的证明 `¬m≤n`，所以 `¬s≤s ¬m≤n` 即可作为 `¬ (suc m ≤ suc n)` 的证明。

当我们写 `_≤ᵇ_` 时，我们必须写两个其他的函数 `≤ᵇ→≤` 和 `≤→≤ᵇ` 来证明其正确性。作为对比，`_≤?_` 的定义

自身就证明了其正确性，由类型即可得知。`_≤?_` 的代码也比 `_≤ᵇ_`、`≤ᵇ→≤` 和 `≤→≤ᵇ` 加起来要简洁的多。我们稍后将会证明，如果我们需要后三者，我们亦可简单地从 `_≤?_` 中派生出来。

我们可以使用我们新的函数来计算出我们之前需要自己想出来的证明。

```
_ : 2 ≤? 4 ≡ yes (s≤s (s≤s z≤n))
_ = refl

_ : 4 ≤? 2 ≡ no (¬s≤s (¬s≤s ¬s≤z))
_ = refl
```

你可以验证 Agda 的确计算出了这些值。输入 `C-c C-n` 并给出 `2 ≤? 4` 或者 `4 ≤? 2` 作为 需要的表达式，Agda 会输出如上的值。

（小细节：如果我们不把 `¬s≤z` 和 `¬s≤s` 作为顶层函数来定义，而是使用内嵌的匿名函数， Agda 可能会在规范化否定的证明中出现问题。）

练习 `_<?_` （推荐）

与上面的函数相似，定义一个判定严格不等性的函数：

```
postulate
  _<?_ : ∀ (m n : ℕ) → Dec (m < n)
```

```
-- 请将代码写在此处。
```

练习 `_≡ℕ?_` （实践）

定义一个函数来判定两个自然数是否相等。

```
postulate
  _≡ℕ?_ : ∀ (m n : ℕ) → Dec (m ≡ n)
```

```
-- 请将代码写在此处。
```

从可判定的值到布尔值，从布尔值到可判定的值

好奇的读者可能会思考能不能重用 `m ≤ᵇ n` 的定义，加上它与 `m ≤ n` 等价的证明， 来证明可判定性。的确，我们是可以做到的：

```
_≤?′_ : ∀ (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
```

如果 `m ≤ᵇ n` 为真，那么 `≤ᵇ→≤` 会返回一个 `m ≤ n` 成立的证明。 如果 `m ≤ᵇ n` 为假，那么 `≤→≤ᵇ` 会取一个 `m ≤ n` 成立的证明，将其转换为一个矛盾。

在这个证明中，`with` 语句的三重约束是必须的。如果我们取而代之的写：

```
_≤?″_ : ∀ (m n : ℕ) → Dec (m ≤ n)
m ≤?″ n with m ≤ᵇ n
... | true  =  yes (≤ᵇ→≤ m n tt)
... | false =  no (≤→≤ᵇ {m} {n})
```

那么 Agda 对于每条语句会有一个抱怨：

```
⊤ !=< (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
```

将表达式放在 `with` 语句中能让 Agda 利用下列事实：当 `m ≤ᵇ n` 为真时，`T (m ≤ᵇ n)` 是 `⊤`；当 `m ≤ᵇ n` 为假时，`T (m ≤ᵇ n)` 是 `⊥`。

然而，总体来说还是直接定义 `_≤?_` 比较方便，正如之前部分中那样。如果有人真的很需要 `_≤ᵇ_`，那么它和它的性质可以简单地从 `_≤?_` 中派生出来，正如我们接下来要展示的一样。

擦除（Erasure）将一个可判定的值转换为一个布尔值：

```
⌊_⌋ : ∀ {A : Set} → Dec A → Bool
⌊ yes x ⌋ = true
⌊ no ¬x ⌋ = false
```

使用擦除，我们可以简单地从 `_≤?_` 中派生出 `_≤ᵇ_`：

```
_≤ᵇ′_ : ℕ → ℕ → Bool
m ≤ᵇ′ n = ⌊ m ≤? n ⌋
```

更进一步来说，如果 `D` 是一个类型为 `Dec A` 的值，那么 `T ⌊ D ⌋` 当且仅当 `A` 成立时成立：

```
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
```

使用这些，我们可以简单地派生出 `T (m ≤ᵇ′ n)` 当且仅当 `m ≤ n` 成立时成立。

```
≤ᵇ′→≤ : ∀ {m n : ℕ} → T (m ≤ᵇ′ n) → m ≤ n
≤ᵇ′→≤ = toWitness

≤→≤ᵇ′ : ∀ {m n : ℕ} → m ≤ n → T (m ≤ᵇ′ n)
≤→≤ᵇ′ = fromWitness
```

总结来说，最好避免直接使用布尔值，而使用可判定的值。如果有需要布尔值的时候，它们和它们的性质可以简单地从对应的可判定的值中派生而来。

大多数读者对于布尔值的逻辑运算符很熟悉了。每个逻辑运算符都可以被延伸至可判定的值。

两个布尔值的合取当两者都为真时为真，当任一为假时为假：

```
infixr 6 _∧_

_∧_ : Bool → Bool → Bool
true ∧ true = true
false ∧ _    = false
_     ∧ false = false
```

在 Emacs 中，第三个等式的左手边显示为灰色，表示这些等式出现的顺序决定了是第二条还是第三条会被匹配到。然而，不管是哪一条被匹配到，结果都是一样的。

相应地，给定两个可判定的命题，我们可以判定它们的合取：

```
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 }
```

两个命题的合取当两者都成立时成立，其否定则当任意一者否定成立时成立。如果两个都成立，我们将每一证明放入数据对中，作为合取的证明。如果任意一者的否定成立，假设整个合取将会引入一个矛盾。

同样地，在 Emacs 中，第三条等式在左手边以灰色显示，说明等式的顺序决定了第二条还是第三条会被匹配。

这一次，我们给出的结果会因为是第二条还是第三条而不一样。如果两个命题都不成立，我们选择第一个来构造矛盾，但选择第二个也是同样正确的。

两个布尔值的析取当任意为真时为真，当两者为假时为假：

```
infixr 5 _∨_

_∨_ : Bool → Bool → Bool
true ∨ _     = true
_    ∨ true  = true
false ∨ false = false
```

在 Emacs 中，第二个等式的左手边显示为灰色，表示这些等式出现的顺序决定了是第一条还是第二条会被匹配到。然而，不管是哪一条被匹配到，结果都是一样的。

相应地，给定两个可判定的命题，我们可以判定它们的析取：

```
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 }
```

两个命题的析取当任意一者成立时成立，其否定则当两者的否定成立时成立。如果任意一者成立，我们使用其证明来作为析取的证明。如果两个的否定都成立，假设任意一者都会引入一个矛盾。

同样地，在 Emacs 中，第二条等式在左手边以灰色显示，说明等式的顺序决定了第一条还是第二条会被匹配。这一次，我们给出的结果会因为是第二条还是第三条而不一样。如果两个命题都成立，我们选择第一个来构造析取，但选择第二个也是同样正确的。

一个布尔值的否定当值为真时为假，反之亦然：

```
not : Bool → Bool
not true  = false
not false = true
```

相应地，给定一个可判定的命题，我们可以判定它的否定：

```
¬? : ∀ {A : Set} → Dec A → Dec (¬ A)
¬? (yes x) = no (¬¬-intro x)
¬? (no ¬x) = yes ¬x
```

我们直接把 yes 和 no 交换。在第一个等式中，右手边断言了 `¬ A` 的否定成立，也就是说 `¬ ¬ A` 成立——这是一个 `A` 成立时可以简单得到的推论。

还有一个与蕴涵相对应，但是稍微不那么知名的运算符：

```
_⊃_ : Bool → Bool → Bool
_     ⊃ true  = true
false ⊃ _     = true
true  ⊃ false = false
```

当任何一个布尔值为真的时候，另一个布尔值恒为真，我们成为第一个布尔值蕴涵第二个布尔值。因此，两者的蕴涵在第二个为真或者第一个为假时为真，在第一个为真而第二个为假时为假。在 Emacs 中，第二个等式的左手边显示为灰色，表示这些等式出现的顺序决定了是第一条还是第二条会被匹配到。然而，不管是哪一条被匹配到，结果都是一样的。

相应地，给定两个可判定的命题，我们可以判定它们的析取：

```
_→-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))
```

两者的蕴涵在第二者成立或者第一者的否定成立时成立，其否定在第一者成立而第二者否定成立时成立。蕴涵成立的证明是一个从第一者成立的证明到第二者成立的证明的函数。如果第二者成立，那么这个函数直接返回第二者的证明。如果第一者的否定成立，那么使用第一者成立的证明，构造一个矛盾。如果第一者成立，第二者不成立，给定蕴涵成立的证明，我们必须构造一个矛盾：我们将成立的证明 `f` 应用于第一者成立的证明 `x`，再加以第二者否定成立的证明 `¬y` 来构造矛盾。

同样地，在 Emacs 中，第二条等式在左手边以灰色显示，说明等式的顺序决定了第一条还是第二条会被匹配。这一次，我们给出的结果会因为是哪一条被匹配而不一样，但两者都是同样正确的。

练习 `erasure`（实践）

证明擦除将对应的布尔值和可判定的值的操作联系了起来：

```
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 ⌋
```

练习 `iff-erasure`（推荐）

给出与同构与嵌入章节中 `_⇔_` 相对应的布尔值与可判定的值的操作，并证明其对应的擦除：

```
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 ⌋
```

```
-- 请将代码写在此处。
```

互映证明

让我们回顾一下章节自然数中 `monus` 的定义。如果从一个较小的数中减去一个较大的数，结果为零。毕竟我们总是要得到一个结果。我们可以用其他方式定义吗？可以定义一版带有守卫（*guarded*）的减法——只有当 `n ≤ m` 时才能从 `m` 中减去 `n` ：

```
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
```

然而这种定义难以使用，因为我们必须显式地为 `n ≤ m` 提供证明：

```
_ : minus 5 3 (s≤s (s≤s (s≤s z≤n))) ≡ 2
_ = refl
```

这个问题没有通用的解决方案，但是在上述的情景下，我们恰好静态地知道这两个数字。这种情况下，我们可以使用一种被称为*互映证明*（*proof by reflection*）的技术。 实质上，在类型检查的时候我们可以让 Agda 运行可判定的等式 `n ≤? m` 并且保证 `n ≤ m` ！

我们使用「隐式参数」的一个特性来实现这个功能。如果 Agda 可以填充一个记录类型的所有字段，那么 Agda 就可以填充此记录类型的隐式参数。 由于空记录类型没有任何字段，Agda 总是会设法填充空记录类型的隐式参数。这就是 `T` 类型被定义成空记录的原因。

这里的技巧是设置一个类型为 `T ⌊ n ≤? m ⌋` 的隐式参数。让我们一步一步阐述这句话的含义。 首先，我们运行判定过程 `n ≤? m` 。它向我们提供了 `n ≤ m` 是否成立的证据。我们擦除证据得到布尔值。 最后，我们应用 `T` 。回想一下，`T` 将布尔值映射到证据的世界：`true` 变成了单位类型 `⊤` ， `false` 变成了空类型 `⊥` 。在操作上，这个类型的隐式参数起到了守卫的作用。

- 如果 `n ≤ m` 成立，隐式参数的类型规约为 `⊤` 。 然后 Agda 会欣然地提供隐式参数。
- 否则，类型规约为 `⊥` ，Agda 无法为此类型提供对应的值，因此会报错。例如，如果我们调用 `3 - 5` 会得到 `_n≤m_254 : ⊥` 。

我们使用之前定义的 `toWitness` 获得了 `n ≤ m` 的证据：

```
_-_ : (m n : ℕ) {n≤m : T ⌊ n ≤? m ⌋} → ℕ
_-_ m n {n≤m} = minus m n (toWitness n≤m)
```

我们现在只要能静态地知道这两个数就可以安全地使用 `_-_` 了：

```
_ : 5 - 3 ≡ 2
_ = refl
```

事实上，这种惯用语法非常普遍。标准库为 `T ⌊ ? ⌋` 定义了叫做 `True` 的同义词：

```
True : ∀ {Q} → Dec Q → Set
True Q = T ⌊ Q ⌋
```

练习 `False`

给出 `True`，`toWitness` 和 `fromWitness` 的相反定义。分别称为 `False`，`toWitnessFalse` 和 `fromWitnessFalse`。

标准库

```
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  逻辑和 (\and, \wedge)
∨  U+2228  逻辑或 (\or, \vee)
⊃  U+2283  超集 (\sup)
ᵇ  U+1D47  修饰符小写 B  (\^b)
⌊  U+230A  左向下取整 (\clL)
```

⌋ U+230B 右向下取整 (\clR)

Chapter 10

Lists: 列表与高阶函数

```
module plfa.part1.Lists where
```

本章节讨论列表（List）数据类型。我们用列表作为例子，来使用我们之前学习的技巧。同时，列表也给我们带来多态类型（Polymorphic Types）和高阶函数（Higher-order Functions）的例子。

导入

```
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 (_≃_; _⇔_)
```

列表

Agda 中的列表如下定义：

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A

infixr 5 _::_
```

我们来仔细研究这个定义。如果 `A` 是个集合，那么 `List A` 也是一个集合。接下来的两行告诉我们 `[]`（读作 *nil*）是一个类型为 `A` 的列表（通常被叫做空列表），`_::_`（读作 *cons*，是 *constructor* 的简写）取一个类型为 `A` 的值，和一个类型为`List A` 的值，返回一个类型为 `List A` 的值。`_::_` 运算符的优先级是 5，向右结合。

例如：

```
_ : List ℕ
_ = 0 :: 1 :: 2 :: []
```

表示了一个三个自然数的列表。因为 `_::_` 向右结合，这一项被解析成`0 :: (1 :: (2 :: []))`。 在这里， `0` 是列表的第一个元素，称之为头（Head），`1 :: (2 :: [])` 是剩下元素的列表，称之为尾（Tail）。列表是一个奇怪的怪兽：它有一头一尾，中间没有东西，然而它的尾巴又是一个列表！

正如我们所见，参数化的类型可以被转换成索引类型。上面的定义与下列等价：

```
data List′ : Set → Set where
  []′  : ∀ {A : Set} → List′ A
  _::′_ : ∀ {A : Set} → A → List′ A → List′ A
```

每个构造子将参数作为隐式参数。因此我们列表的例子也可以写作：

```
_ : List ℕ
_ = _::_ {ℕ} 0 (_::_ {ℕ} 1 (_::_ {ℕ} 2 ([] {ℕ})))
```

此处我们将隐式参数显式地声明。

包含下面的编译器指令

```
{-# BUILTIN LIST List #-}
```

告诉 Agda，`List` 类型对应了 Haskell 的列表类型，构造子`[]` 和 `_::_` 分别代表了 nil 和 cons，这可以让列表的表示更加的有效率。

列表语法

我们可以用下面的定义，更简便地表示列表：

```
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 ∷ []
```

这是我们第一次使用模式声明。举例来说，第三行告诉我们`[ x , y , z ]`等价于`x ∷ y ∷ z ∷ []`。前者可以在模式或者等式的左手边，或者是等式右手边的项中出现。

附加

我们对于列表的第一个函数写作`_++_`，读作附加（Append）：

```
infixr 5 _++_

_++_ : ∀ {A : Set} → List A → List A → List A
[] ++ ys       = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)
```

`A`类型是附加的隐式参数，这让这个函数变为一个**多态（Polymorphic）**函数（即可以用作多种类型）。一个列表附加到空列表会得到该列表本身；一个列表附加到非空列表所得到的列表，其头与附加到的非空列表相同，尾与所附加的列表相同。

我们举个例子，来展示将两个列表附加的计算过程：

```
_ : [ 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 ∷ []
```

```
  ∎
```

附加两个列表需要对于第一个列表元素个数线性的时间。

论证附加

我们可以与用论证数几乎相同的方法来论证列表。下面是附加满足结合律的证明：

```
++-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)
  ∎
```

证明对于第一个参数进行归纳。起始步骤将列表实例化为`[]`，由直接的运算可证。 归纳步骤将列表实例化为`x :: xs`，由直接的运算配合归纳假设可证。与往常一样，归纳假设由递归使用证明函数来表明，此处为`++-assoc xs ys zs`。

回忆到 Agda 支持片段。使用`cong (x ::_)`可以将归纳假设：

```
(xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
```

提升至等式：

```
x :: ((xs ++ ys) ++ zs) ≡ x :: (xs ++ (ys ++ zs))
```

即证明中所需。

我们也可以简单地证明 `[]` 是 `_++_` 的左幺元和右幺元。 左幺元的证明从定义中即可得：

```
++-identityˡ : ∀ {A : Set} (xs : List A) → [] ++ xs ≡ xs
++-identityˡ xs =
  begin
    [] ++ xs
  ≡⟨⟩
    xs
  ∎
```

右幺元的证明可由简单的归纳得到：

```
++-identityʳ : ∀ {A : Set} (xs : List A) → xs ++ [] ≡ xs
++-identityʳ [] =
  begin
    [] ++ []
  ≡⟨⟩
    []
  ∎
++-identityʳ (x ∷ xs) =
  begin
    (x ∷ xs) ++ []
  ≡⟨⟩
    x ∷ (xs ++ [])
  ≡⟨ cong (x ∷_) (++-identityʳ xs) ⟩
    x ∷ xs
  ∎
```

我们之后会了解到，这三条性质表明了 `_++_` 和 `[]` 在列表上构成了一个*幺半群*（Monoid）。

长度

在下一个函数里，我们来寻找列表的长度：

```
length : ∀ {A : Set} → List A → ℕ
length []       = zero
length (x ∷ xs) = suc (length xs)
```

同样，它取一个隐式参数 `A`。空列表的长度为零。非空列表的长度比其尾列表长度多一。

我们用下面的例子来展示如何计算列表的长度：

```
_ : 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))
  ∎
```

计算列表的长度需要关于列表元素个数线性的时间。

在倒数第二行中，我们不可以直接写 `length []`，而需要写 `length {ℕ} []`。因为 `[]` 没有元素，Agda 没有足够的信息来推导其隐式参数。

论证长度

两个附加在一起的列表的长度是两列表长度之和：

```
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
  ∎
```

证明对于第一个参数进行归纳。起始步骤将列表实例化为 `[]`，由直接的运算可证。 如同之前一样，Agda 无法推导 `length` 的隐式参数，所以我们必须显式地给出这个参数。归纳步骤将列表实例化为 `x ∷ xs`，由直接的运算配合归纳假设可证。与往常一样，归纳假设由递归使用证明函数来表明，此处为 `length-++ xs ys`， 由 `cong suc` 来提升。

反转

我们可以使用附加，来简单地构造一个函数来反转一个列表：

```
reverse : ∀ {A : Set} → List A → List A
reverse []       = []
reverse (x ∷ xs) = reverse xs ++ [ x ]
```

空列表的反转是空列表。非空列表的反转是其头元素构成的单元列表附加至其尾列表反转之后的结果。

下面的例子展示了如何反转一个列表。

```
_ : 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 ]
  ∎
```

这样子反转一个列表需要列表长度二**次**的时间。这是因为反转一个长度为`n`的列表需要将长度为`1`、`2`直到`n - 1`的列表附加起来，而附加两个列表需要第一个列表长度线性的时间，因此加起来就需要`n * (n - 1) / 2`的时间。（我们将在本章节后部分验证这一结果）

练习`reverse-++-distrib`（推荐）

证明一个列表附加到另外一个列表的反转即是反转后的第二个列表附加至反转后的第一个列表：

```
reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
```

练习`reverse-involutive`（推荐）

当一个函数应用两次后与恒等函数作用相同，那么这个函数是一个**对合**（Involution）。证明反转是一个对合：

```
reverse (reverse xs) ≡ xs
```

更快地反转

上面的定义虽然论证起来方便，但是它比期望中的实现更低效，因为它的运行时间是关于列表长度的二次函数。我们可以将反转进行推广，使用一个额外的参数：

```
shunt : ∀ {A : Set} → List A → List A → List A
shunt [] ys       = ys
shunt (x :: xs) ys = shunt xs (x :: ys)
```

这个定义对于第一个参数进行递归。第二个参数会变_大_，但这样做没有问题，因为我们递归的参数在变_小_。

转移（Shunt）与反转的关系如下：

```
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
  ∎
```

证明对于第一个参数进行归纳。起始步骤将列表实例化为`[]`，由直接的运算可证。 归纳步骤将列表实例化为`x ∷ xs`，由归纳假设和附加的结合律可证。当我们使用归纳假设时，第二个参数实际上变**大**了，但是这样做没有问题，因为我们归纳的参数变**小**了。

使用一个会在归纳或递归的参数变小时，变大的辅助参数来进行推广，是一个常用的技巧。这个技巧在以后的证明中很有用。

在定义了推广的转移之后，我们可以将其特化，作为一个更高效的反转的定义：

```
reverse′ : ∀ {A : Set} → List A → List A
reverse′ xs = shunt xs []
```

因为我们之前证明的引理，我们可以直接地证明两个定义是等价的：

```
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
```

```
  ∎
```

下面的例子展示了如何快速反转列表 `[ 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 ∷ []
  ∎
```

现在反转一个列表需要的时间与列表的长度线性相关。

映射

映射将一个函数应用于列表中的所有元素，生成一个对应的列表。映射是一个**高阶函数**（Higher-Order Function）的例子，它取一个函数作为参数，返回一个函数作为结果：

```
map : ∀ {A B : Set} → (A → B) → List A → List B
map f []       = []
map f (x ∷ xs) = f x ∷ map f xs
```

空列表的映射是空列表。非空列表的映射生成一个列表，其头元素是原列表的头元素在应用函数之后的结果，其尾列表是原列表的尾列表映射后的结果。

下面的例子展示了如何使用映射来增加列表中的每一个元素：

```
_ : 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 ∷ []
  ∎
```

映射需要关于列表长度线性的时间。

我们常常可以利用柯里化，将映射作用于一个函数，获得另一个函数，然后在之后的时候应用获得的函数：

```
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 ]
  ∎
```

任何对于另外一个类型参数化的类型，例如列表，都有对应的映射，其接受一个函数，并返回另一个从由给定函数定义域参数化的类型，到由给定函数值域参数化的函数。除此之外，一个对于 *n* 个类型 参数化的类型会有一个对于 *n* 个函数参数化的映射。

练习 `map-compose`（实践）

证明函数组合的映射是两个映射的组合：

```
map (g ∘ f) ≡ map g ∘ map f
```

证明的最后一步需要外延性。

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

练习 `map-++-distribute`（实践）

证明下列关于映射与附加的关系：

```
map f (xs ++ ys) ≡ map f xs ++ map f ys
```

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

练习 `map-Tree`（实践）

定义一个树数据类型，其叶节点类型为 `A`，内部节点类型为 `B`：

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  node : Tree A B → B → Tree A B → Tree A B
```

定义一个对于树的映射运算符：

```
map-Tree : ∀ {A B C D : Set} → (A → C) → (B → D) → Tree A B → Tree C D
```

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

折叠

折叠取一个运算符和一个值，并使用运算符将列表中的元素合并至一个值，如果给定的列表为空，则使用给定的值：

```
foldr : ∀ {A B : Set} → (A → B → B) → B → List A → B
foldr _⊗_ e []       = e
foldr _⊗_ e (x :: xs) = x ⊗ foldr _⊗_ e xs
```

空列表的折叠是给定的值。非空列表的折叠使用给定的运算符，将头元素和尾列表的折叠合并起来。

下面的例子展示了如何使用折叠来对一个列表求和：

```
_ : 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)))
∎
```

折叠需要关于列表长度线性的时间。

我们常常可以利用柯里化，将折叠作用于一个运算符和一个值，获得另一个函数，然后在之后的时候应用获得的函数：

```
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
  ∎
```

正如列表由两个构造子 `[]` 和 `_∷_`，折叠函数取两个参数 `e` 和 `_⊗_`（除去列表参数）。推广来说，一个有 *n* 个构造子的数据类型，会有对应的 取 *n* 个参数的折叠函数。

举另外一个例子，观察

```
foldr _∷_ [] xs ≡ xs
```

若 `xs` 的类型为 `List A`，那么我们就会有一个 `foldr` 的实例，其中的 `A` 为 `A`，而 `B` 为 `List A`。它遵循

```
xs ++ ys ≡ foldr _∷_ ys xs
```

二者相等的证明留作练习。

练习 `product` （推荐）

使用折叠来定义一个计算列表数字之积的函数。例如：

```
product [ 1 , 2 , 3 , 4 ] ≡ 24
```

```
-- 请将代码写在此处。
```

练习 `foldr-++` （推荐）

证明折叠和附加有如下的关系：

```
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
```

练习 `map-is-foldr`

证明映射可以用折叠定义：

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

此证明需要外延性。

练习 `map-is-foldr` （实践）

请证明 map 可使用 fold 来定义：

```
map f ≡ foldr (λ x xs → f x ∷ xs) []
```

此证明需要外延性。

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

练习 `fold-Tree`（实践）

请为预先给定的三个类型定义一个合适的折叠函数：

```
fold-Tree : ∀ {A B C : Set} → (A → C) → (C → B → C → C) → Tree A B → C
```

```
-- 请将代码写在此处。
```

练习 `map-is-fold-Tree`（实践）

对于树数据类型，证明与 `map-is-foldr` 相似的性质。

```
-- 请将代码写在此处。
```

证明 `sum-downFrom`（延伸）

定义一个向下数数的函数：

```
downFrom : ℕ → List ℕ
downFrom zero    = []
downFrom (suc n) = n ∷ downFrom n
```

例如：

```
_ : downFrom 3 ≡ [ 2 , 1 , 0 ]
_ = refl
```

证明数列之和 `(n - 1) + ⋯ + 0` 等于 `n * (n ∸ 1) / 2`：

```
sum (downFrom n) * 2 ≡ n * (n ∸ 1)
```

幺半群

一般来说，我们会对于折叠函数使用一个满足结合律的运算符，和这个运算符的左右幺元。这意味着这个运算符和这个值形成了一个**幺半群（Monoid）**。

我们可以用一个合适的记录类型来定义幺半群：

```
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
```

举例来说，加法和零，乘法和一，附加和空列表，都是幺半群：

```
+-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
    }
```

如果 `_⊗_` 和 `e` 构成一个幺半群，那么我们可以用相同的运算符和一个任意的值来表示折叠：

```
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
  ∎
```

在之前的练习中，我们证明了以下定理：

```
postulate
  foldr-++ : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) (xs ys : List A) →
    foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ (foldr _⊗_ e ys) xs
```

使用之前练习中的一个结论，我们可以得到如下推论：

```
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
  ∎
```

练习 `foldl`（实践）

定义一个函数 `foldl`，与 `foldr` 相似，但是运算符向左结合，而不是向右。例如：

```
foldr _⊗_ e [ x , y , z ]  =  x ⊗ (y ⊗ (z ⊗ e))
foldl _⊗_ e [ x , y , z ]  =  ((e ⊗ x) ⊗ y) ⊗ z
```

```
-- 请将代码写在此处。
```

练习 `foldr-monoid-foldl`（实践）

证明如果 `_⊗_` 和 `e` 构成幺半群，那么 `foldr _⊗_ e` 和 `foldl _⊗_ e` 的结果永远是相同的。

```
-- 请将代码写在此处。
```

所有

我们也可以定义关于列表的谓词。最重要的两个谓词是 `All` 和 `Any`。

谓词 `All P` 当列表里的所有元素满足 `P` 时成立：

```
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)
```

这个类型有两个构造子，使用了与列表构造子相同的名称。第一个断言了 `P` 对于空列表的任何元素成立。 第二个断言了如果 `P` 对于列表的头元素和尾列表的所有元素成立，那么 `P` 对于这个列表的任何元素成立。 Agda 使用类型来区分构造子是用于构造一个列表，还是构造 `All P` 成立的证明。

比如说，`All (_≤ 2)` 对于一个每一个元素都小于等于二的列表成立。 回忆 `z≤n` 证明了对于任意 `n`，`zero ≤ n` 成立； 对于任意 `m` 和 `n`，如果 `m≤n` 证明了 `m ≤ n`，那么 `s≤s m≤n` 证明了 `suc m ≤ suc n`：

```
_ : All (_≤ 2) [ 0 , 1 , 2 ]
_ = z≤n :: s≤s z≤n :: s≤s (s≤s z≤n) :: []
```

这里 `_::_` 和 `[]` 是 `All P` 的构造子，而不是 `List A` 的。 这三项分别是 `0 ≤ 2`、`1 ≤ 2` 和 `2 ≤ 2` 的证明。

（读者可能会思考诸如 `[_,_,_]` 的模式是否可以用于构造 `All` 类型的值， 像构造 `List` 类型的一样，因为

两者使用了相同的构造子。事实上这样做是可以的，只要两个类型在模式声明时在作用域内。然而现在不是这样的情况，因为 `List` 先于 `[_,_,_]` 定义，而 `All` 在之后定义。）

任意

谓词 `Any P` 当列表里的一些元素满足 `P` 时成立：

```
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)
```

第一个构造子证明了列表的头元素满足`P`，第二个构造子证明的列表的尾列表中的一些元素满足 `P`。举例来说，我们可以如下定义列表的成员关系：

```
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)
```

比如说，零是列表 `[ 0 , 1 , 0 , 2 ]` 中的一个元素。我们可以用两种方法来展示这个事实，对应零在列表中出现了两次：第一个元素和第三个元素：

```
_ : 0 ∈ [ 0 , 1 , 0 , 2 ]
_ = here refl

_ : 0 ∈ [ 0 , 1 , 0 , 2 ]
_ = there (there (here refl))
```

除此之外，我们可以展示三不在列表之中，因为任何它在列表中的证明会推导出矛盾：

```
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 ()))))
```

`()` 出现了五次，分别表示没有 `3 ≡ 0`、`3 ≡ 1`、 `3 ≡ 0`、 `3 ≡ 2` 和`3 ∈ []` 的证明。

所有和附加

一个谓词对两个附加在一起的列表的每个元素都成立，当且仅当这个谓词对两个列表的每个元素都成立：

```
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 ⟩
```

练习 `Any-++-⇔` （推荐）

使用 `Any` 代替 `All` 与一个合适的 `_×_` 的替代，证明一个类似于 `All-++-⇔` 的结果。 作为结论，展示关联 `_∈_` 和 `_++_` 的一个等价关系。

```
-- 请将代码写在此处。
```

练习 `All-++-≃` （延伸）

证明 `All-++-⇔` 的等价关系可以被扩展至一个同构关系。

```
-- 请将代码写在此处。
```

练习 `¬Any⇔All¬` （推荐）

请证明 `Any` 和 `All` 满足一个版本的德摩根定律：

```
(¬_ ∘ Any P) xs ⇔ All (¬_ ∘ P) xs
```

（你能明白为什么这里的 `_∘_` 被泛化到任意层级很重要吗？如全体多态一节所述。）

以下定律是否也成立？

```
(¬_ ∘ All P) xs ⇔ Any (¬_ ∘ P) xs
```

若成立，请证明；否则请解释原因。

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

练习 `¬Any≃All¬`（拓展）

请证明等价的 `¬Any⇔All¬` 可以被扩展成一个同构。你需要使用外延性。

```
-- 请将代码写在此处
```

练习 `All-∀`（实践）

请证明 `All P xs` 同构于 `∀ x → x ∈ xs → P x`.

```
-- 请将代码写在此处
```

练习 `Any-∃`（实践）

请证明 `Any P xs` 同构于`∃[ x ] (x ∈ xs × P x)`.

```
-- 请将代码写在此处
```

如果成立，请证明；如果不成立，请解释原因。

所有的可判定性

如果我们将一个谓词看作一个返回布尔值的函数，那么我们可以简单的定义一个类似于 `All` 的函数，其当给定谓词对于列表每个元素返回真时返回真：

```
all : ∀ {A : Set} → (A → Bool) → List A → Bool
all p = foldr _∧_ true ∘ map p
```

我们可以使用高阶函数 `map` 和 `foldr` 来简洁地写出这个函数。

正如所希望的那样，如果我们将布尔值替换成可判定值，这与 `All` 是相似的。首先，回到将 `P` 当作一个类型为 `A → Set` 的函数的概念，将一个类型为 `A` 的值 `x` 转换成 `P x` 对 `x` 成立 的证明。我们成 `P` 为**可判定的**（Decidable），如果我们有一个函数，其在给定 `x` 时能够判定 `P x`：

```
Decidable : ∀ {A : Set} → (A → Set) → Set
Decidable {A} P = ∀ (x : A) → Dec (P x)
```

那么当谓词 `P` 可判定时，我们亦可判定列表中的每一个元素是否满足这个谓词：

```
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 }
```

如果列表为空，那么 `P` 显然对列表的每个元素成立。否则，证明的结构与两个可判定的命题是可判定的证明相似，不过我们使用 `_∷_` 而不是 `⟨_,_⟩` 来整合头元素和尾列表的证明。

练习 `Any?`（扩展）

正如 `All` 有类似的 `all` 和 `All?` 形式，来判断列表的每个元素是否满足给定的谓词，那么 `Any` 也有类似的 `any` 和 `Any?` 形式，来判断列表的一些元素是否满足给定的谓词。给出它们的定义。

```
-- 请将代码写在此处。
```

练习 `split`（扩展）

关系 `merge` 在两个列表合并的结果为给定的第三个列表时成立。

```
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)
```

例如

```
_ : merge [ 1 , 4 ] [ 2 , 3 ] [ 1 , 2 , 3 , 4 ]
_ = left-∥ (right-∥ (right-∥ (left-∥ [])))
```

给定一个可判定谓词和一个列表，我们可以将该列表拆分成两个列表，二者可以合并成原列表，其中一个列表的所有元素都满足该谓词，而另一个列表中的所有元素都不满足该谓词。

在列表上定义一个传统 `filter` 函数的变体，如下所示，它接受一个可判定谓词和一个列表，返回一个所有元素都满足该谓词的列表，和一个所有元素都不满足的列表，以及与它们相应的证明。

```
split : ∀ {A : Set} {P : A → Set} (P? : Decidable P) (zs : List A)
  → ∃[ xs ] ∃[ ys ] ( merge xs ys zs × All P xs × All (¬_ ∘ P) ys )
```

```
-- 请将代码写在此处
```

标准库

标准库中可以找到与本章节中相似的定义：

```
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)
```

标准库中的 `IsMonoid` 与给出的定义不同，因为它可以针对特定的等价关系参数化。

`Relation.Unary` 和 `Relation.Binary` 都定义了 `Decidable` 的某个版本，一个 用于单元关系（正如本章中的单元谓词 `P`），一个用于二元关系（正如之前使用的 `_≤_`）。

Unicode

本章使用了下列 Unicode：

```
∷  U+2237  比例  (\::)
⊗  U+2297  带圈的乘号  (\otimes, \ox)
∈  U+2208  元素属于  (\in)
∉  U+2209  元素不属于  (\inn, \notin)
```

Part II

第二分册：编程语言基础

Chapter 11

Lambda: λ-演算简介

```
module plfa.part2.Lambda where
```

λ-演算，最早由逻辑学家 Alonzo Church 发表，是一种只含有三种构造的演算——变量（Variable）、抽象（Abstraction）与应用（Application）。**λ-演算**刻画了**函数抽象**（Functional Abstract）的核心概念。这样的概念以函数、过程和方法的形式，在基本上每一个编程语言中都有体现。 简单类型的 λ-演算 （Simply-Typed Lambda Calculus，简写为 STLC）是 λ-演算的一种变体，由 Church 在 1940 年发表。除去之前提到的三种构造，简单类型的 λ-演算还拥有函数类型和任何所需的基本类型。 Church 使用了最简单的没有任何操作的基本类型。 我们在这里使用 Plotkin 的**可编程的可计算函数**（Programmable Computable Functions，PCF），并加入自然数和递归函数及其相关操作。

在本章中，我们将形式化简单类型的 λ-演算，给出它的语法、小步语义和类型规则。 在下一章 Properties 中，我们将证明它的主要性质，包括可进性与保型性。 后续的章节将研究 λ-演算的不同变体。

请注意，我们在这里使用的方法**不是**将它形式化的推荐方法。使用 de Bruijn 索引和 固有类型的项（我们会在 DeBruijn 章节中进一步研究），可以让我们的形式化更简洁。不过，我们先从使用带名字的变量和外在类型的项来表示 λ-演算开始。一方面是因为这样表述的项更易于阅读，另一方面是因为这样的表述更加传统。

这一章启发自《软件基础》（*Software Foundations*）/《程序语言基础》（*Programming Language Foundations*）中对应的 *Stlc* 的内容。我们的不同之处在于使用显式的方法来表示上下文（由标识符和类型的序对组成的列表），而不是偏映射（从标识符到类型的偏函数）。 这样的做法与后续的 de Bruijn 索引表示方法能更好的对应。我们使用自然数作为基础类型，而不是布尔值，这样我们可以表示更复杂的例子。特别的是，我们将可以证明（两次！）二加二得四。

导入

```
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)
```

项的语法

项由七种构造组成。首先是 λ-演算中核心的三个构造：

- 变量 `` ` x ``
- 抽象 `ƛ x ⇒ N`
- 应用 `L · M`

三个与自然数有关的构造：

- 零 `` `zero ``
- 后继 `` `suc ``
- 匹配 `case L [zero⇒ M |suc x ⇒ N ]`

一个与递归有关的构造：

- 不动点 `μ x ⇒ M`

抽象也被叫做 *λ-抽象*，这也是 λ-演算名字的由来。

除了变量和不动点以外，每一个项要么构造了一个给定类型的值（抽象产生了函数，零和后继产生了自然数），要么析构了一个这样的值（应用使用了函数，匹配使用了自然数）。我们在给项赋予类型的时候将重新探讨这一对应关系。构造子对应了引入规则，析构子对应了消去规则。

下面是以 Backus-Naur 范式（BNF）给出的语法：

```
L, M, N  ::=
  ` x  |  ƛ x ⇒ N  |  L · M  |
  `zero  |  `suc M  |  case L [zero⇒ M |suc x ⇒ N ]  |
  μ x ⇒ M
```

而下面是用 Agda 形式化后的代码：

```
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
```

我们用字符串来表示标识符。 我们使用的优先级使得 λ-抽象和不动点结合的最不紧密，其次是应用，再是后继，结合得最紧密的是变量的构造子。 匹配表达式自带了括号。

项的例子

下面是一些项的例子：自然数二、一个将自然数相加的函数和一个计算二加二的项：

```
two : Term
two = `suc `suc `zero

plus : Term
plus = μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
        case ` "m"
          [zero⇒ ` "n"
          |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

加法的递归定义与我们一开始在 Naturals 章节中定义的 `_+_` 相似。 在这里，变量「m」被约束了两次，一个在 λ-抽象中，另一次在匹配表达式的后继分支中。第一次使用的「m」指代前者，第二次使用的指代后者。任何在后继分支中的「m」必须指代后者，因此我们称之为后者**屏蔽**（Shadow）了前者。后面我们会证实二加二得四，也就是说下面的项

```
plus · two · two
```

会规约为 `` `suc `suc `suc `suc `zero `` 。

第二个例子里，我们使用高阶函数来表示自然数。 具体来说，数字 *n* 由一个接受两个参数的函数来表示，这个函数将第一个参数 应用于第二个参数上 *n* 次。 这样的表示方法叫做自然数的 **Church 表示法**。下面是一个项的例子：Church 表示法的数字二、一个将两个用 Church 表示法表示的数字相加的函数、 一个计算后继的函数和一个计算二加二的项：

```
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")
```

Church 法表示的二取两个参数 `s` 和 `z`，将 `s` 应用于 `z` 两次。 加法取两个数 `m` 和 `n`，函数 `s` 和参数 `z`，使用 `m` 将 `s` 应用于 使用 `n` 应用于 `s` 和 `z` 的结果。因此 `s` 对于 `z` 被应用了 `m` 加 `n` 次。为了方便起见，我们定义一个计算后继的函数。 为了将一个 Church 数转化为对应的自然数，我们将它应用到 `sucᶜ` 函数和自然数零上。同样，我们之后会证明二加二得四，也就是说，下面的项

```
plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

会规约为 `` `suc `suc `suc `suc `zero `` 。

练习 `mul` （推荐）

写出一个项来定义两个自然数的乘法。你可以使用之前定义的 `plus` 。

```
-- 请将代码写在此处。
```

练习 `mulᶜ` （习题）

写出一个项来定义两个用 Church 法表示的自然数的乘法。你可以使用之前定义的 `plusᶜ` 。（当然也可以不用，用或不使都有很好的表示方法）

```
-- 请将代码写在此处。
```

练习 `primed` （延伸）

用 `` ` "x" `` 而不是 `x` 来表示变量可能并不是每个人都喜欢。我们可以加入下面的定义，来帮助我们表示项的例子：

```
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 : ⊥
```

我们希望只在两个参数不相等的时候应用这个函数；我们引入一个空类型 `⊥` 的项 `impossible` 作为公设，用来表示第二种情况不会发生。如果我们使用 C-c C-n 来范式化这个项

```
ƛ′ two ⇒ two
```

Agda 会警告我们出现了不可能的情况。⊥-elim (plfa.part2.Lambda.impossible (`suc (`suc `zero)) (`suc (`suc `zero)))

假设一件不可能的事情是一个有用的方法，但是我们必须加以注意。因为这样的假设能让我们构造出**任何**命题，不论真假。

现在我们可以用下面的形式重新写出 `plus` 的定义：

```
plus′ : Term
plus′ = μ′ + ⇒ ƛ′ m ⇒ ƛ′ n ⇒
          case′ m
            [zero⇒ n
            |suc m ⇒ `suc (+ · m · n) ]
  where
  + = ` "+"
  m = ` "m"
  n = ` "n"
```

用这样的形式写出乘法的定义。

形式化与非正式

在形式化语义的非正式表述中，我们使用变量名来消除歧义，用 `x` 而不是 `` ` x `` 来表示一个变量项。Agda 要求我们对两者进行区分。

相似地来说，非正式的表达在**对象语言**（Object Language，我们正在描述的语言）和**元语言**（Meta-Language，我们用来描述对象语言的语言）中使用相同的记法来表示函数类型、λ-抽象和函数应用，相信读者可以通过上下文区分两种语言。而 Agda 并不能做到这样，因此我们在目标语言中使用 `ƛ x ⇒ N` 和 `L · M`，与我们使

用的元语言 Agda 中的 `λ x → N` 和 `L M` 相对。

约束变量与自由变量

在抽象 `ƛ x ⇒ N` 中，我们把 `x` 叫做**约束变量**，`N` 叫做**抽象体**。λ-演算一个重要的特性是将相同的约束变量同时重命名不会改变一个项的意义。因此下面的五个项

- `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 "😇" ⇒ ` "😈" · (` "😈" · ` "😇")`

都可以认为是等价的。使用 Haskell Curry 引入的约定，这样的规则 用希腊字母 `α` （*alpha*） 来表示，因此这样的等价关系也叫做 **α-重命名**。

当我们从一个项中观察它的子项时，被约束的变量可能会变成自由变量。考虑下面的项：

- `ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")` `s` 和 `z` 都是约束变量。
- `ƛ "z" ⇒ ` "s" · (` "s" · ` "z")` `z` 是约束变量，`s` 是自由变量。
- `` ` "s" · (` "s" · ` "z") `` `s` 和 `z` 都是自由变量。

我们将没有自由变量的项叫做**闭项**，否则它是一个**开项**。上面的三个项中，第一个是闭项，剩下两个是开项。我们在讨论规约时，会注重闭项。

一个变量在不同地方出现时，可以同时是约束变量和自由变量。在下面的项中：

```
(ƛ "x" ⇒ ` "x") · ` "x"
```

内部的 `x` 是约束变量，外部的是自由变量。使用 α-重命名，上面的项等同于

```
(ƛ "y" ⇒ ` "y") · ` "x"
```

在此之中 `y` 是约束变量，`x` 是自由变量。**Barendregt 约定**，一个常见的约定，使用 α-重命名来保证约束变量与自由变量完全不同。这样可以避免因为约束变量和自由变量名称相同而造成的混乱。

匹配和递归同样引入了约束变量，我们也可以使用 α-重命名。下面的项

```
μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
  case ` "m"
    [zero⇒ ` "n"
    |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

注意这个项包括了两个 `m` 的不同绑定，第一次出现在第一行，第二次出现在最后一行。这个项与下面的项等同

```
μ "plus" ⇒ ƛ "x" ⇒ ƛ "y" ⇒
  case ` "x"
    [zero⇒ ` "y"
    |suc "x′" ⇒ `suc (` "plus" · ` "x′" · ` "y") ]
```

其中两次出现的 `m` 现在用 `x` 和 `x′` 两个不同的名字表示。

值

值（Value）是一个对应着答案的项。因此，`` `suc `suc `suc `suc `zero `` 是一个值，而 `plus · two · two` 不是。根据惯例，我们将所有的抽象当作值；所以 `plus` 本身是一个值。

谓词 `Value M` 当一个项 `M` 是一个值时成立：

```
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)
```

后续文中，我们用 `V` 和 `W` 来表示值。

正式与非正式

在形式化语义的非正式表达中，我们用元变量 `V` 来表示一个值。在 Agda 中，我们必须使用 `Value` 谓词来显式地表达。

其他方法

另一种定义不注重封闭的项，将变量视作值。`ƛ x ⇒ N` 只有在 `N` 是一个值的时候，才是一个值。这是 Agda 标准化项的方法，我们在 Untyped 章节中考虑这种方法。

替换

λ-演算的核心操作是将一个项中的变量用另一个项来替换。替换在定义函数应用的操作语义中起到了重要的作用。 比如说，我们有

```
  (ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) · sucᶜ · `zero
⟶
  (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero
⟶
  sucᶜ · (sucᶜ · `zero)
```

其中，我们在抽象体中用 `sucᶜ` 替换 `` ` "s" ``，用 `` `zero `` 替换 `` ` "z" `` 。

我们将替换写作 `N [ x ≔ V ]`，意为用 `V` 来替换项 `N` 中出现的所有自由变量 `x`。 简短地说，就是用`V` 来替换 `N` 中的 `x`，或者是把 `N` 中的`x` 换成 `V`。 替换只在 `V` 是一个封闭项时有效。它不一定是一个值，我们在这里使用 `V` 是因为常常我们使用值进行替换。

下面是一些例子：

- `(ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) ["s" ≔ sucᶜ]` 得 `ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")`。
- `(sucᶜ · (sucᶜ · ` "z")) ["z" ≔ `zero ]` 得 `sucᶜ · (sucᶜ · `zero)`。
- `(ƛ "x" ⇒ ` "y") ["y" ≔ `zero ]` 得 `ƛ "x" ⇒ `zero`。
- `(ƛ "x" ⇒ ` "x") ["x" ≔ `zero ]` 得 `ƛ "x" ⇒ ` "x"`。
- `(ƛ "y" ⇒ ` "y") ["x" ≔ `zero ]` 得 `ƛ "y" ⇒ ` "y"`。

在倒数第二个例子中，用 `` `zero `` 在`ƛ "x" ⇒ ` "x"` 出现的 `x` 得到的**不是** `ƛ "x" ⇒ `zero`，因为`x` 是抽象中的约束变量。 约束变量的名称是无关的，`ƛ "x" ⇒ ` "x"` 和`ƛ "y" ⇒ ` "y"` 都是恒等函数。 可以认为`x` 在抽象体内和抽象体外是**不同的**变量，而它们恰好拥有一样的名字。

我们将要给出替换的定义在用来替换变量的项是封闭时有效。这是因为用**不**封闭的项可能需要对于约束变量进行重命名。例如：

- `(ƛ "x" ⇒ ` "x" · ` "y") [ "y" ≔ ` "x" · `zero]` 不应该得到 `(ƛ "x" ⇒ ` "x" · (` "x" · `zero))`.

不同如上，我们应该将约束变量进行重命名，来防止捕获：

- `(ƛ "x" ⇒ ` "x" · ` "y") [ "y" ≔ ` "x" · `zero ]` 应该得到 `ƛ "x′" ⇒ ` "x′" · (` "x" · `zero)`.

这里的 `x′` 是一个新的、不同于 `x` 的变量。带有重命名的替换的形式化定义更加复杂。在这里，我们将替换限制在封闭的项之内，可以避免重命名的问题，但对于我们要做的后续的内容来说也是足够的。

下面是对于封闭项替换的 Agda 定义：

```
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 ]
```

下面我们来看一看前三个情况：

- 对于变量，我们将需要替换的变量 `y` 与项中的变量 `x` 进行比较。 如果它们相同，我们返回 `V`，否则返回 `x` 不变。
- 对于抽象，我们将需要替换的变量 `y` 与抽象中的约束变量 `x` 进行比较。如果它们相同，我们返回抽象不变，否则对于抽象体内部进行替换。
- 对于应用，我们递归地替换函数和其参数。

匹配表达式和递归也有约束变量，我们使用与抽象相似的方法处理它们。除此之外的情况，我们递归地对于子项进行替换。

例子

下面是上述替换正确性的证明：

```
_ : (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
```

小测验

下面替换的结束是?

```
(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))`

练习 `_[_⦂=_]′` (延伸)

上面的替换定义中有三条语句(`ƛ`、`case` 和 `μ`)使用了 `with` 语句来处理约束变量。将上述语句的共同部分提取成一个函数,然后用共同递归重写替换的定义。

```
-- 请将代码写在此处。
```

规约

我们接下来给出 λ-演算的传值规约规则。规约一个应用时,我们首先规约左手边,直到它变成一个值(必须是抽象);接下来我们规约后手边,直到它变成一个值;最后我们使用替换,把变量替换成参数。

在非正式的操作语言表达中,我们可以如下写出应用的规约规则:

```
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 ]
```

稍后给出的 Agda 版本的规则与上述相似，但是我们需要将全称量化显式地表示出来，也需要使用谓词来表示一个是值的项。

规则可以分为两类。 兼容性规则让我们规约一个项的一部分。我们用希腊字母 `ξ` （*xi*）开头的规则表示。当一个项规约到足够的时候，它将会包括一个构造子和一个析构子，在这里是 `ƛ` 和 `·`， 我们可以直接规约。这样的规则我们用希腊字母 `β` （*beta*）表示，也被称为 **β-规则**。

一些额外的术语：可以匹配规约规则左手边的项被称之为**可规约项**（Redex）。在可规约项 `(ƛ x ⇒ N) · V` 中，我们把 `x` 叫做函数的**形式参数**（形参，Formal Parameter）， 把 `V` 叫做函数应用的**实际参数**（实参，Actual Parameter）。β-规约将形参用实参来替换。

如果一个项已经是一个值，它就没有可以规约的规则；反过来说，如果一个项可以被规约，那么它就不是一个值。我们在下一章里证明这概括了所有的情况——所以良类型的项要么可以规约要么是一个值。

对于数字来说，零不可以规约，后继可以对它的子项进行规约。匹配表达式先将它的参数规约至一个数字，然后根据它是零还是后继选择相应的分支。不动点会把约束变量替换成整个不动点项——这是我们唯一一处用项、而不是值进行的替换。

我们用下面的形式在 Agda 里形式化这些规则：

```
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 ]
```

我们小心地设计这些规约规则，使得一个项的子项在整项被规约之前先被规约。这被称为**传值**（Call-by-value）规约。

除此之外，我们规定规约的顺序是从左向右的。这意味着规约是**确定的**（Deterministic）：对于任何一个项，最多存在一个可以被规约至的项。换句话说，我们的规约关系 `⟶` 实际上是一个函数。

这种解释一个项的含义的方法叫做**小步操作语义**（Small-step Operational Semantics）。 如果 `M ⟶ N`，我们称之为项 `M` **规约** 至项 `N`，也称之为项 `M` **步进**至（Step to） 项 `N`。每条兼容性规则以另一条规约规则作为前提；因此每一步都会用到一条 β-规则，用零或多条兼容性规则进行调整。

小测验

下面的项步进至哪一项?

```
(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")`

下面的项步进至哪一项?

```
(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")`

下面的项步进至哪一项？（`twoᶜ` 和 `sucᶜ` 如之前的定义）

```
twoᶜ · sucᶜ · `zero  ⟶  ???
```

1. `sucᶜ · (sucᶜ · `zero)`
2. `(ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero`
3. `` `zero ``

自反传递闭包

步进并不是故事的全部。总的来说，对于一个封闭的项，我们想要对它反复地步进，直到规约至一个值。这样可以用定义步进关系 `⟶` 的自反传递闭包 `⟶↠` 来完成。

我们以一个零或多步的步进关系的序列来定义这样的自反传递闭包，这样的形式与 Equality 章节中的等式链论证形式相似：

```
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
```

我们如下理解这个关系：

- 对于项 `M`，我们可以一步也不规约而得到类型为 `M ⟶↠ M` 的步骤，写作 `M ∎`。

- 对于项 `L`，我们可以使用 `L ⟶ M` 类型步进一步，再使用 `M ⟶↠ N` 类型步进零或多步，得到类型为 `L ⟶↠ N` 的步骤，写作 `L ⟶⟨ L⟶M ⟩ M⟶↠N`。其中，`L⟶M` 和 `M⟶↠N` 是相应类型的步骤。

在下一部分我们可以看到，这样的记法可以让我们用清晰的步骤来表示规约的例子。

我们也可以用包括 `⟶` 的最小的自反传递关系作为另一种定义：

```
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
```

这样的三个构造子分别表示了 `⟶↠′` 包括 `⟶`、自反和传递的性质。证明两者是等价的是一个很好的练习。（的确，一者嵌入了另一者）

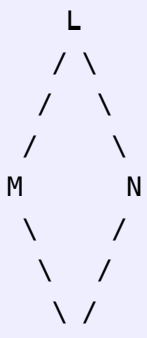
练习 `—↠≲—↠′` （习题）

证明自反传递闭包的第一种记法嵌入了第二种记法。 为什么它们不是同构的？

```
-- 请将代码写在此处。
```

合流性

在讨论规约关系时，有一个重要的性质是**合流性**（Confluence）。如果项 `L` 规约至两个项 `M` 和项 `N`，那么它们都可以规约至同一个项 `P`。我们可以用下面的图来展示这个性质：



```
          P
```

图中，`L`、`M` 和 `N` 由全称量词涵盖，而 `P` 由存在量词涵盖。如果图中的每条线代表了零或多步规约步骤，这样的性质被成为合流性。如果上面的两条线代表一步规约步骤，下面的两条线代表零或多步规约步骤，这样的性质被成为菱形性质。用符号表示为：

```
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))
```

在本章中我们讨论的规约系统是确定的。用符号表示为：

```
postulate
  deterministic : ∀ {L M N}
    → L —→ M
    → L —→ N
      ------
    → M ≡ N
```

我们可以简单地证明任何确定的规约关系满足菱形性质，任何满足菱形性质的规约关系满足合流性。因为，我们研究的规则系统平凡地满足了合流性。

例子

我们用一个简单的例子开始。Church 数二应用于后继函数和零可以得到自然数二：

```
_ : 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)
∎
```

下面的例子中我们规约二加二至四：

```
_ : 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)))
  ∎
```

我们用 Church 数规约同样的例子：

```
_ : 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)))
  ∎
```

下一章节中，我们研究如何计算这样的规约序列。

练习 `plus-example` （习题）

使用规约序列，证明一加一得二。

```
-- 请将代码写在此处。
```

类型的语法

我们只有两种类型：

- 函数：`A ⇒ B`
- 自然数：`` `ℕ ``

和之前一样，我们需要使用与 Agda 不一样的名称来防止混淆。

下面是类型的 BNF 形式语法：

```
A, B, C  ::=  A ⇒ B | `ℕ
```

下面是用 Agda 的形式化：

```
infixr 7 _⇒_

data Type : Set where
  _⇒_ : Type → Type → Type
  `ℕ : Type
```

优先级

与 Agda 中一致，两个或多个参数的函数以科里化的形式表示。以右结合的方式定义 `_⇒_`、左结合的方式定义 `_'_` 更加方面。 因此：

- `(`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ` 表示 `((`ℕ ⇒ `ℕ) ⇒ (`ℕ ⇒ `ℕ))`。
- `plus · two · two` 表示 `(plus · two) · two`。

小测验

- 下面给出的项的类型是什么？

  ```
  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 ``

在适当的情况下，可以给出多于一个答案。

- 下面给出的项的类型是什么？
  `` (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 ``

在适当的情况下，可以给出多于一个答案。

赋型

上下文

在规约时，我们只讨论封闭的项，但是在赋型时，我们必须考虑带有自由变量的项。给一个项赋型时，我们必须先给它的子项赋型。而在给一个抽象的抽象体赋型时，抽象的约束变量在抽象体内部是自由的。

上下文（Context）将变量和类型联系在一起。 我们用 `Γ` 和 `Δ` 来表示上下文。 我们用 `∅` 表示空的上下文，用 `Γ , x ⦂ A` 表示扩充 `Γ` ，将变量 `x` 对应至类型 `A` 。 例如：

- `` ∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ``

这个上下文将变量 `"s"` 对应至类型 `` `ℕ ⇒ `ℕ `` ， 将变量 `"z"` 对应至类型 `` `ℕ `` 。

上下文如下形式化：

```
infixl 5 _,_⦂_

data Context : Set where
  ∅ : Context
  _,_⦂_ : Context → Id → Type → Context
```

练习 `Context-≃` （习题）

证明 `Context` 与 `List (Id × Type)` 同构。

例如，如下的上下文

```
∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ
```

和如下的列表相关。

```
[ ⟨ "z" , `ℕ ⟩ , ⟨ "s" , `ℕ ⇒ `ℕ ⟩ ]
```

```
-- 请将代码写在此处。
```

查询判断

我们使用两种**判断**。第一种写作

```
Γ ∋ x ⦂ A
```

表示在上下文 `Γ` 中变量 `x` 的类型是 `A`。这样的判断叫做**查询**（Lookup）判断。 例如，

- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "s" ⦂ `ℕ ⇒ `ℕ`

分别给出了变量 `"z"` 和 `"s"` 对应的类型。 我们使用符号 `∋` （读作 “ni”，反写的 “in”），因为 `Γ ∋ x ⦂ A` 与查询 `x ⦂ A` 是否在与 `Γ` 对应的列表中存在相似。

如果上下文中有相同名称的两个变量，那么查询会返回被约束的最近的变量，它**遮盖**（Shadow）了另一个变量。例如：

- `∅ , "x" ⦂ `ℕ ⇒ `ℕ , "x" ⦂ `ℕ ∋ "x" ⦂ `ℕ`.

在这里 `"x" ⦂ `ℕ ⇒ `ℕ` 被 `"x" ⦂ `ℕ` 遮盖了。

我们如下形式化查询：

```
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
```

构造子 `Z` 和 `S` 大致与列表包含关系 `_∈_` 的 `here` 和 `there` 构造子对应。 但是构造子 `S` 多取一个参数，来保证查询时我们不会查询一个被**遮盖**的同名变量。

用 `S` 构造子会比较麻烦，因为每次都需要提供 `x ≢ y` 的证明。例如：

```
_ : ∅ , "x" ⦂ `ℕ ⇒ `ℕ , "y" ⦂ `ℕ , "z" ⦂ `ℕ ∋ "x" ⦂ `ℕ ⇒ `ℕ
_ = S (λ()) (S (λ()) Z)
```

取而代之的是，我们在类型检查时可以使用以互映证明来检查不等性的「智慧构造子」：

```
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
```

赋型判断

第二种判断写作

```
Γ ⊢ M ⦂ A
```

表示在上下文 `Γ` 中，项 `M` 有类型 `A`。 上下文`Γ` 为 `M` 中的所有自由变量提供了类型。 例如：

- `∅ , "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`

赋型可以如下形式化：

```
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
```

赋型规则由对应的项的构造子来命名。

大多数规则有第二个名字，从逻辑中的惯例得到。规则的名称也可以用类型的连接符中得到，引入和消去连接符分别用 `-I` 和 `-E` 表示。我们从上往下阅读时，引入和消去的规则一目了然：前者**引入**了一个带有连接符的式子，其出现在结论中，而不是条件中；后者**消去**了带有连接符的式子，其出现在条件中，而不是结论中。引入规则表示了如何构造一个给定类型的值（抽象产生函数、零和后继产生自然数），而消去规则表示了如何析构一

个给定类型的值（应用使用函数，匹配表达式使用自然数）。

另外需要注意的是有三处地方（`⊢ƛ`、`⊢case` 和 `⊢μ`），上下文被 `x` 和相应的类型所扩充，对应着三处约束变量的引入。

这些规则是确定的，对于每一项至多有一条规则使用。

类型推导的例子

类型推导对应着树。在非正式的记法中，下面是 Church 数二的类型推导：

```
                      ∋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
```

其中 `∋s` 和 `∋z` 是下面两个推导的简写：

```
            -------------- Z
"s" ≠ "z"   Γ₁ ∋ "s" ⦂ A ⇒ A
--------------------------- S      ------------ Z
Γ₂ ∋ "s" ⦂ A ⇒ A                   Γ₂ ∋ "z" ⦂ A
```

其中`Γ₁ = Γ , "s" ⦂ A ⇒ A`、`Γ₂ = Γ , "s" ⦂ A ⇒ A , "z" ⦂ A`。给出的推导对于任意的 `Γ` 和 `A` 有效，例如， 我们可以取`Γ` 为 `∅` 和 `A` 为 `` `ℕ ``。

上面的推导可以如下用 Agda 形式化：

```
Ch : Type → Type
Ch A = (A → A) → A → A

⊢two : ∀ {Γ A} → Γ ⊢ two ⦂ Ch A
⊢two = ⊢ƛ (⊢ƛ (⊢` ∋s · (⊢` ∋s · ⊢` ∋z)))
  where
  ∋s = S′ Z
  ∋z = Z
```

下面是针对二加二的赋型：

```
⊢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
```

与之前的例子不同，我们以任意上下文，而不是空上下文来赋型。这让我们能够在其他除了顶层之外的上下文中使用这个推导。 这里的查询判断 `∋m` 和 `∋m′` 指代两个变量 `"m"` 的绑定。作为对比，查询判断 `∋n` 和 `∋n′` 指代同一个变量 `"n"` 的绑定，但是查询的上下文不同， 第一次 `"n"` 出现在在上下文的最后，第二次在 `"m"` 之后。

对 Church 数赋型的余下推导如下：

```
⊢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
```

与 Agda 交互

可以交互式地构造类型推导。 从声明开始：

```
⊢sucᶜ : ∅ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
⊢sucᶜ = ?
```

使用 C-c C-l 让 Agda 创建一个洞，并且告诉我们期望的类型：

```
⊢sucᶜ = { }0
?0 ⦂ ∅ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
```

现在使用 C-c C-r 来填补这个洞。Agda 注意到 `sucᶜ` 最外层的项是 `ƛ`，应该使用 `⊢ƛ` 来赋型。 `⊢ƛ` 规则需要一个变量，用一个新的洞表示：

```
⊢sucᶜ = ⊢ƛ { }1
?1 ⦂ ∅ , "n" ⦂ `ℕ ⊢ `suc ` "n" ⦂ `ℕ
```

再次使用 C-c C-r 来填补洞：

```
⊢sucᶜ = ⊢ƛ (⊢suc { }2)
?2 ⦂ ∅ , "n" ⦂ `ℕ ⊢ ` "n" ⦂ `ℕ
```

再来一次：

```
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` { }3))
?3 ⦂ ∅ , "n" ⦂ `ℕ ∋ "n" ⦂ `ℕ
```

再次尝试使用 C-c C-r 得到下面的消息：

```
Don't know which constructor to introduce of Z or S
```

我们使用填入 `Z` 。如果我们使用 C-c C-space，Agda 证实我们完成了：

```
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` Z))
```

我们也可以使用 C-c C-a，用 Agsy 来自动完成。

在 Inference 章节中，我们会展示如何使用 Agda 来直接计算出类型推导。

查询是单射

查询关系 `Γ ∋ x ⦂ A` 是一个单射。 对于所有的 `Γ` 和 `x`， 至多有一个 `A` 满足这个判断：

```
∋-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′
```

赋型关系 `Γ ⊢ M ⦂ A` 不是一个单射。例如，在任何 `Γ` 中项 `ƛ "x" ⇒ ` "x"` 有类型`A ⇒ A`，`A` 为任何类型。

非例子

我们也可以证明一些项**不是**可赋型的。例如，我们接下来证明项 `` `zero · `suc `zero `` 是不可赋型的。 原因在于我们需要使得 `` `zero `` 既是一个函数又是一个自然数。

```
nope₁ : ∀ {A} → ¬ (∅ ⊢ `zero · `suc `zero ⦂ A)
nope₁ (() · _)
```

第二个例子，我们证明项`ƛ "x" ⇒ ` "x" · ` "x"` 是不可赋型的。 原因在于我们需要满足 `A ⇒ B ≡ A` 的两个类型`A` 和 `B`：

```
nope₂ : ∀ {A} → ¬ (∅ ⊢ ƛ "x" ⇒ ` "x" · ` "x" ⦂ A)
nope₂ (⊢ƛ (⊢` ∋x · ⊢` ∋x′)) = contradiction (∋-injective ∋x ∋x′)
  where
  contradiction : ∀ {A B} → ¬ (A ⇒ B ≡ A)
  contradiction ()
```

小测验

对于下面的每一条，如果可以推导，给出类型`A`，否则说明为什么这样的 `A` 不存在。

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`

对于下面的每一条，如果可以推导，给出类型 `A`、`B` 和`C`，否则说明为什么这样的类型 不存在。

1. `∅ , "x" ⦂ A ⊢ ` "x" · ` "x" ⦂ B`
2. `∅ , "x" ⦂ A , "y" ⦂ B ⊢ ƛ "z" ⇒ ` "x" · (` "y" · ` "z") ⦂ C`

练习 `⊢mul` （推荐）

使用你之前写出的项 `mul`，给出其良类型的推导。

```
-- 请将代码写在此处。
```

练习 `⊢mulᶜ` （习题）

使用你之前写出的项 `mulᶜ`，给出其良类型的推导。

```
-- 请将代码写在此处。
```

Unicode

本章中使用了以下 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
```

我们用短划 `—` 和箭头 `→` 来构造规约 `—→`。自反传递闭包 `—↠` 也类似。

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 ⦂ ∀ {Γ 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`ᶜ and `suc`ᶜ 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 correspon-

dences 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 ✗ ✗ 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

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
```

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:

```
_ : two꜀ · suc꜀ · `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+2083  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-caseᵿ
caseᵿ L [injᵢ x ⇒ M |injⱼ y ⇒ N ] ⟶ caseᵿ L′ [injᵢ x ⇒ M |injⱼ y ⇒ N ]

------------------------------------------------------- β-injᵢ
caseᵿ (`injᵢ V) [injᵢ x ⇒ M |injⱼ y ⇒ N ] ⟶ M [ x ≔ V ]

------------------------------------------------------- β-injⱼ
caseᵿ (`injⱼ W) [injᵢ x ⇒ M |injⱼ y ⇒ N ] ⟶ N [ y ≔ W ]
```

Example

Here is a function to swap the components of a sum:

```
swapᵿ : ∀ {A B : Type} → A `ᵿ B → B `ᵿ A
swapᵿ = ƛ z ⇒ caseᵿ z
                [injᵢ x ⇒ `injⱼ x
                |injⱼ y ⇒ `injᵢ 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
  `T                                 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
----------------------- caseT or ⊤-E
Γ ⊢ caseT L [tt⇒ M ] ⦂ A
```

Reduction

```
L ⟶ L′
------------------------------------ ξ-caseT
caseT L [tt⇒ M ] ⟶ caseT L′ [tt⇒ M ]

---------------------- β-caseT
caseT `tt [tt⇒ M ] ⟶ M
```

Example

Here is half the isomorphism between `A` and `A `× `⊤` rewritten in the new notation:

```
from×⊤-case ı ∅ ⊢ A `× `⊤ ⇒ A
from×⊤-case = ƛ z ⇒ case× z
                     [⟨ x , y ⟩⇒ caseT y
                                  [tt⇒ x ] ]
```

Translation

We can translate the alternative formulation into one without case:

```
(caseT 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 ∋ x ⦂ A )
    ----------------------------
  → ¬ ∃[ A ]( Γ , y ⦂ B ∋ 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 ∋ 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 ∋ 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 ∋ x ⦂ A` holds, or its negation:

```
lookup : ∀ (Γ : Context) (x : Id)
         ------------------------
       → Dec (∃[ A ]( Γ ∋ 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
        (⊢↑
          (⊢`
            (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 ∅ (twoᶜ ↓ `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 twoᶜ ↓ `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
           (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:

```
    L
   / \
  /   \
 /     \
M₁      M₂
 \     /
  \   /
   \ /
    N
```

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 x₁ x x)((λ x₁ x) a) ⟶ (λ x₁ x x) a
(λ x₁ x x)((λ x₁ x) a) ⟶ ((λ x₁ x) a) ((λ x₁ 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

```
   M
  /|
```

```
  / |
 /  |
N   2
 \  |
  \ |
   \|
    M⁺
```

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

```
    M
   / \
  1   *
 /     \
N       N′
 \     /
  *   1
   \ /
    L
```

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:

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_;
         #_; twoᶜ; 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 ⊢ (⊥ ⦂ 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, `𝒞 M ≃ 𝒞 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.

```
𝒟 (` x) ≃ ...
𝒟 (ƛ M) ≃ ... 𝒟 M ...
𝒟 (M · N) ≃ ... 𝒟 M ... 𝒟 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 × ⌊ 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 $\wp(\omega)$ (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 $\wp(\omega)$, 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 $\mathscr{F}$, 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 $\mathscr{F}$ has three parameters, the denotation `D` of the subterm `M`, an environment `γ`, and a value `v`. If we define $\mathscr{F}$ 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 $\mathscr{F}$ 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 $\mathscr{F}$, we hope to prove that

```
ℰ (ƛ N) ≃ ℱ (ℰ N)
```

The function $\mathscr{F}$ 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
```

```
    ------------
  → 𝒯 (ℰ N) γ u
sub-𝒯 d ⊑-bot = tt
sub-𝒯 d (⊑-fun lt lt′) = sub (up-env d lt) lt′
sub-𝒯 d (⊑-conj-L lt lt₁) = ⟨ sub-𝒯 d lt , sub-𝒯 d lt₁ ⟩
sub-𝒯 d (⊑-conj-R1 lt) = sub-𝒯 (proj₁ d) lt
sub-𝒯 d (⊑-conj-R2 lt) = sub-𝒯 (proj₂ d) lt
sub-𝒯 {v = v₁ ↦ v₂ ⊔ v₁ ↦ v₃} {v₁ ↦ (v₂ ⊔ v₃)} ⟨ N2 , N3 ⟩ ⊑-dist =
  ⊔-intro N2 N3
sub-𝒯 d (⊑-trans x₁ x₂) = sub-𝒯 (sub-𝒯 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-𝒯` in the case for the `sub` rule.

```
ℰ→𝒯ℰ : ∀{Γ}{γ : Env Γ}{N : Γ , ⋆ ⊢ ⋆}{v : Value}
  → ℰ (ƛ N) γ v
    ------------
  → 𝒯 (ℰ N) γ v
ℰ→𝒯ℰ (↦-intro d) = d
ℰ→𝒯ℰ ⊥-intro = tt
ℰ→𝒯ℰ (⊔-intro d₁ d₂) = ⟨ ℰ→𝒯ℰ d₁ , ℰ→𝒯ℰ d₂ ⟩
ℰ→𝒯ℰ (sub d lt) = sub-𝒯 (ℰ→𝒯ℰ 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→𝒯ℰ{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.

```
𝒯ℰ→ℰ : ∀{Γ}{γ : Env Γ}{N : Γ , ⋆ ⊢ ⋆}{v : Value}
  → 𝒯 (ℰ N) γ v
    ------------
  → ℰ (ƛ N) γ v
𝒯ℰ→ℰ {v = ⊥} d = ⊥-intro
𝒯ℰ→ℰ {v = v₁ ↦ v₂} d = ↦-intro d
𝒯ℰ→ℰ {v = v₁ ⊔ v₂} ⟨ d1 , d2 ⟩ = ⊔-intro (𝒯ℰ→ℰ d1) (𝒯ℰ→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 = ⟨ ℰƛ→𝒻ℰ , 𝒻ℰ→ℰƛ ⟩
```

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.

```
ℰ·→●ℰ : ∀{Γ}{γ : Env Γ}{L M : Γ ⊢ ⋆}{v : Value}
  → ℰ (L · M) γ v
    ----------------
  → (ℰ L ● ℰ M) γ v
ℰ·→●ℰ (↦-elim{v = v′} d₁ d₂) = inj₂ ⟨ v′ , ⟨ d₁ , d₂ ⟩ ⟩
ℰ·→●ℰ {v = ⊥} ⊥-intro = inj₁ ⊑-bot
```

```
ℰ'→●ℰ {Γ}{γ}{L}{M}{v} (⊔-intro{v = v₁}{w = v₂} d₁ d₂)
   with ℰ'→●ℰ d₁ | ℰ'→●ℰ 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↓⊔ ⟩ ⟩
ℰ'→●ℰ {Γ}{γ}{L}{M}{v} (sub d lt)
   with ℰ'→●ℰ 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 `(ℰ L ● ℰ M) γ v`.
- In case `⊥-intro` we have `v = ⊥`. We conclude that `v ⊑ ⊥`.
- In case `⊔-intro` we have `ℰ (L , M) γ v₁` and `ℰ (L , M) γ v₂` and need to show `(ℰ L ● ℰ M) γ (v₁ ⊔ v₂)`. By the induction hypothesis, we have `(ℰ L ● ℰ M) γ v₁` and `(ℰ L ● ℰ 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 `ℰ L ● ℰ 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 `𝒞 L ≃ 𝒞 L′` and `𝒞 M ≃ 𝒞 M′` imply `𝒞 (L · M) ≃ 𝒞 (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 ⊢ ★}
  → 𝒞 L ≃ 𝒞 L′
  → 𝒞 M ≃ 𝒞 M′
    ------------------------
  → 𝒞 (L · M) ≃ 𝒞 (L′ · M′)
app-cong {Γ}{L}{L′}{M}{M′} L≅L′ M≅M′ =
  start
    𝒞 (L · M)
  ≃⟨ app-equiv ⟩
    𝒞 L ● 𝒞 M
  ≃⟨ ●-cong L≅L′ M≅M′ ⟩
    𝒞 L′ ● 𝒞 M′
  ≃⟨ ≃-sym app-equiv ⟩
    𝒞 (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 ⊢ ⋆}
  → 𝒞 M ≃ 𝒞 N
    --------------------------
  → 𝒞 (plug C M) ≃ 𝒞 (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 `𝒻` and apply `●` functions in the cases for lambda and application, respectively.

```
⟦_⟧ : ∀{Γ} → (M : Γ ⊢ ⋆) → Denotation Γ
⟦ ` x ⟧ γ v = v ⊑ γ x
⟦ ƛ N ⟧ = 𝒻 ⟦ 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 `𝒻` 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 ⟩
    𝒻 (ℰ N)
  ≃⟨ 𝒻-cong (ℰ≃⟦⟧ {M = N}) ⟩
    𝒻 ⟦ 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 —↠ ↑ N  implies  𝒞 L ≃ 𝒞 (↑ 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 γ ⊢ 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 ⌜ ⊆⊔⊔ 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—↠ƛN′ ⟩ =
  let ℰCM≃ℰƛN′ = soundness CM—↠ƛN′ in
  let ℰCM≃ℰCN = compositionality{Γ = Γ}{Δ = ∅}{C = C} ℰM≃ℰN in
  let ℰCN≃ℰƛN′ = ≃-trans (≃-sym ℰCM≃ℰCN) ℰCM≃ℰƛN′ in
    cbn→reduce (proj₂ (proj₂ (proj₂ (↓→⇓ ℰCN≃ℰƛN′))))
```

The proof is direct. Because `plug C —↠ plug C (ƛN′)`, we can apply soundness to obtain

```
ℰ (plug C M) ≃ ℰ (ƛN′)
```

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 (ƛN′).
```

We then apply `↓→⇓` from Chapter Adequacy to deduce

```
∅' ⊢ plug C N ⇓ clos (ƛ 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

附录

附录 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 [x₁:=N] [y₁:=L] = M [y₁:=L] [x₁:= N[y₁:=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)
  ≡⟨⟩
```

```
    ↑ ⟪ exts ids ⟫ N
  ≡⟨ cong ↑_ (cong-sub{M = N} exts-ids refl) ⟩
    ↑ ⟪ ids ⟫ N
  ≡⟨ cong ↑_ sub-id ⟩
    ↑ 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

  ```
  ƛ ⟪ exts σ₁ ⨟ exts σ₂ ⟫ N ≡ ƛ ⟪ 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

后记

附录 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

附录 C

Fonts

```
module plfa.backmatter.Fonts where
```

Preferably, all vertical bars should line up.

```
--------------------------|
abcdefghijklmnopqrstuvwxyz|
ABCDEFGHIJKLMNOPQRSTUVWXYZ|
abcdefghijklmnop rstuvwxyz|
AB DE GHIJKLMNOP R TUVW   |
a   e  hijklmnop rstu  x  |
--------------------------|
----------|
0123456789|
0123456789|
0123456789|
----------|
------------------------|
αβγδεζηθικλμνξοπρστυφχψω|
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ|
------------------------|
----|
≠≠≠≠|
ηηµµ|
ΓΓΔΔ|
ΣΣΠΠ|
λλλλ|
ǺǺǺǺ|
''''|
××××|
ℓℓℓℓ|
```

```
≡≡≡≡|
¬¬¬¬|
≤≤≥≥|
øøøø|
————|
††‡‡|
^^^^|
'''"|
``~~|
⊎⊎⊃⊃|
∧∧∨∨|
⊗⊗⊗⊗|
⊔⊔⊔⊔|
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|
----------|
```