Philip Wadler, Wen Kokke, and Jeremy G. Siek

PROGRAMMING LANGUAGE FOUNDATIONS IN Agda

2021

Contents

Dedication: □□

□ Wanda

□□□□□□□

□ □ □

Philip □ Wanda

□□□□□□□

□ □ □

…

Preface: □□

□□□□□□□□□□□□□□□□□□□□□□□□□□Propositions as Types□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□ □□□□□□□□□□□□□□□□□□□□□□□

□□ □□□□□□□□□□□□□□ Agda □□□□□□□□□□□Specification□ □□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□ λ-□□□Simply-Typed Lambda Calculus□□□ STLC□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□ Agda □□□□□□□Literal Script□□□□□□□ □□ □□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□Literal Programming□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□Donald Knuth□□□□□□□□□□□□□□□□

□□ □□□□□□□□□□□□□□

□□□□

□ 2013 □□□ □□□□□□□□□□□ Benjamin Pierce □□□ TAPL□□□□□□□□□ Pierce □□□□□□□□□□□□□□□ Software Foundations□□□□□ Pierce □□□□□□□□ Coq □□□□□□ Pierce □ ICFP □□□□□ Lambda, The Ultimate TA □□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□ Coq □□□□□□□□□□□□ □□Tactic□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□Product Data Type□□□□ □□□□□□□□□□□□□□□□□□Conjunction□□□□□□Introduction□□□□□□Elimination□□□□□ Coq □□□□□□□□□□Induction Hypothesis□□□□□□□□□□□□□ `notation` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□ `subst N x M` □ `N [x := M]` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□ Agda □□□□□□□□ Agda □□□□□□□□□□□□□□□ □□ □□□ □□□□□□□□□□□□□□□□□□□□□ `_[_:=_]` □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□ Stump □ Verified Functional Programming in Agda □□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Coq □□□ Agda□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□

□□□□□□□□□□□□□ Wen Kokke □□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□ 2018 □□□□□□□□□□□□□

—— Philip Wadler□□□□□□□2018 □ 1 □ - 6 □

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□

GitHub □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Philip Wadler□

[illegible]

[illegible]Agda [illegible]Programming Language Foundations in Agda[illegible]

[illegible] **PLFA-zh** [illegible]

[illegible]

[illegible]

[illegible] PLFA[illegible]

- Stack
- Git
- Agda
- Agda [illegible]
- PLFA

PLFA [illegible] Agda [illegible] Agda [illegible] PLFA[illegible]

Agda [illegible] Agda [illegible] Homebrew [illegible] Debian apt[illegible]Agda [illegible]Agda [illegible] GitHub [illegible] Agda [illegible] Agda [illegible]

macOS [illegible] XCode [illegible]

[illegible] macOS [illegible] XCode [illegible] macOS [illegible]

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

[illegible] Haskell [illegible] Stack

Agda [illegible] Haskell [illegible] Haskell [illegible] Stack [illegible] Stack[illegible] Agda[illegible] Stack [illegible] Haskell [illegible]

- **UNIX** [illegible] **macOS** [illegible] Stack [illegible] macOS [illegible] Homebrew [illegible] Debian [illegible] APT [illegible]haskell-stack[illegible] Stack [illegible]Stack [illegible] `HOME/.local/bin` [illegible] PATH [illegible] shell [illegible] `HOME/.bash_profile` [illegible]

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

□□□□□□□□□□□□□ Stack□□□□

```
stack upgrade
```

- **Windows** □□□Stack □□□□□ Windows □□□□

□□ Git

□□□□□□□□□ Git□□□□ Git □□□□□

□ Stack □□ Agda

□□ □□ Agda □□□□□□□□□□ Stack□□□□□ GitHub □□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□ Zip □□

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

□□□ Agda□□□□□□□□□□□ Stack□

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

□□□□□□□□□□□□□□□□□□□

□□□□□□ GHC

Stack □□□□□□□□□ Glasgow Haskell □□□□GHC□□ □□□□□□□□□□□ GHC □□□□ Stack □□□□□□□□□□□ GHC □□□□□□□ `--system-ghc` □□□□□□□□ `stack-*.yaml` □□□□□□□□□□□ GHC 8.2.2□□□□□

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

□□ Agda □□□□□□□□

□□□□□□□□□□□ Agda □□□□□□□□ □□□□□□ `hello.agda` □□□□□□□□□□□□□

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

□□□□□□□□□ `hello.agda` □□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□

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

□□ PLFA □ Agda □□□

□□□□□ GitHub □□□□□□□□□□□ Zip □□□□□□□□□□□□□□□□Agda □□□□□□

```
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 □□□□□□□ Agda □□□□□□□□□□□□□□□□ `--recurse-submodule` □□□□□ `standard-library` □□□□□□□□ Agda □□□□

□□□□□□□□ `--recurse-submodules` □□□□□□□□□□□□□□□□□

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

□□□□ Zip □□□□ PLFA□□□□□ GitHub □□□□□□□ Agda □□□□□□ □□□□□□□□□□□□□□□□□□□□ Zip □□

```
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.
```

□□□□□□□□ Agda □□□□□□□□□□ □□□□□□□□□□□□□□ □□ `standard-library.agda-lib` □□□□□□□□□□□□□□□□□□ □□□□ `AGDA_DIR` □□□□□□□□□□ UNIX □ macOS □□□ `AGDA_DIR` □□□ `~/.agda` □□ Windows □□□ `AGDA_DIR` □□□□□ `%AppData%\agda` □□ `%AppData%` □□□ `C:\Users\USERNAME\AppData\Roaming` □

- □□ `AGDA_DIR` □□□□□□□□□□□
- □ `AGDA_DIR` □□□□□□□□□□□ `libraries` □□□□ `/path/to/standard-library.agda-lib` □□□□□□□□□□□□ □□□□□ Agda □□□□□□□ `standard-library` □□□□□
- □ `AGDA_DIR` □□□□□□□□□□□ `defaults` □□□□□ `standard-library` □□□□

□□□□□□□□□□□□□□□□ Agda □□□□□□□□

PLFA □□□□□□□□ Agda □□ □□□□□□ `courses` □□□□□□□□□□□□□□□□□□ □□□□□ PLFA □□□ Agda □□□□□□□□□□ `plfa.agda-lib` □□□□□□□□□□□□□□□ `AGDA_DIR/libraries` □□□ `plfa` □□□□□□□□□□ `AGDA_DIR/defaults` □

□□ **Agda** □□□□□□□□□

□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□ `nats.agda` □□□□□□□□□□□

```
open import Data.Nat

ten : ℕ
ten = 10
```

□□□ ℕ □□□ Unicode □□□□□□□□□□ N□□□□□□□□□□□□□□□

□□□□□□□ `nats.agda` □□□□□□□□□□

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

□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□

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

□□ Agda □□□□

Emacs

□□□ Agda □□□□ Emacs□□□ Emacs □□□□□□□

- **UNIX** □□□□□□□ Emacs □□□□□□□□□□□□□GNU Emacs □□□□□□□□□□□□
- **macOS** □□□□□ Emacs □ Aquamacs□□□ GNU Emacs □□□□□ Homebrew □□ MacPorts □□□□□ GNU Emacs □□□□□□□□□
- **Windows** □□□□□ GNU Emacs □□□□□□□□□

□□□□□□□□□□□□□□□□□GNU Emacs □□□□ Emacs □□□□□□□□□ Emacs □□□□□□

Agda □□□ Emacs □□□□□□□□□□□□ Agda□□□□□□□□□□□□ Emacs□

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

□□□□□□ Emacs □□□□□□□□□□□□□□□□ `setup` □□□□□ `.emacs` □□□□□□□□□□□□□□□□□□□

□□ `agda-mode` □Agda □□□ `agda-mode` □ □□□□□□□□□□ Agda□□□□□□□□□□□□□□□□□□□ `agda-mode` □□

□□ `agda-mode` □□□□□

□□□□□□□ `nats.agda` □□□□□ `C-c C-l` □□□□□□□□□□□□□□

□ **Emacs** □□□□□ `agda-mode`

□□□ 2.6.0 □□□Agda □□ Markdown □□□□□□□□□□□□ `.lagda.md` □□□□ □□□□□□□□□□□□□□□□□□□□□□□ Markdown □□□□□ □□□□ `agda-mode` □□□□ `.agda` □ `.lagda.md` □□□□□□□ □□□□□□□□□ Emacs □□□□□

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

□□□□□□□□□□□ `auto-mode-alist` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Emacs Lisp □□□□□□ Emacs □□□□□□□□ `~/.emacs` □ `~/.emacs.d/init.el`□□□ Aquamacs □□□□□□□□□□□□□□ `~/Library/Preferences/Aquamacs Emacs/Preferences` □ `Preferences.el` □□□□ □□ Windows □□□□□□ GNU Emacs □□ □□□□□□□□□□□□

□□□□ Emacs □□□ Mononoki □□

Agda □□□□□□□□□□□ Unicode □□□□□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Unicode □□□□□□□□□□ Mononoki□ □□□□□□□□□ Source Code Pro□ DejaVu Sans Mono □ FreeMono□

□□□□□□ GitHub □□□□□ Mononoki□□□□□□□□□□□□□□□□□□□□□□□□□□ `.otf` □□ `.ttf` □□□ □□□□□□□□□□□□ Mononoki □□□□□□□□□□□□□□ □□□macOS □ Homebrew □ `cask-fonts` cask □□□□ `font-mononiki` □□Debian □ APT □□□ `fonts-mononoki` □□ □□□□□□□□ Emacs □□□□□□□□ Mononoki □□□ Emacs □□□□□□

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

□ Emacs □□□ `agda-mode`

□□□□□□□□□□□□□□□□□□□ `C-c C-l`□

Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□ `C-c C-l` □□□□□Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- `C-c C-c` : □□□□□□□□□□case□
- `C-c C-`□□ □□□
- `C-c C-r` □□□□□□□□□refine□
- `C-c C-a` □□□□□□automatic□
- `C-c C-,` □□□□□□□□□□
- `C-c C-.` □□□□□□□□□□□□□□□□□□□

□□□□□□ emacs-mode □□□

□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□□

- □□ Agda □□□□ `C-c C-l`□
- □ `C-x 1` □□□□□□ Agda □□□
- □ `C-x 3` □□□□□□□□
- □□□□□□□□□□□
- □ `C-x b` □□□□Agda information□□□□□□□□□□

□□□Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□ Emacs □□□ `agda-mode` □□ Unicode □□

□□□ Agda □□□□□□□□□□□□□□□Emacs □□Agda-mode□□□□□□□□□□□□□□□□□□□□□□□□Emacs □□ Agda □□□□□□□□□□□□

□□□□□□□□ `.agda` □□□□□□□□□ □□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□……

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

□□□□□□□□□□□□□□□ ∀ □□□□□□□□□□□□□ `\all` □□□□□□□□□□□□□□□Emacs □□□□□□□□□□……

```
{- I am excited to type ∀
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□ `\->` □□□ `\-` □□□□□……

```
{- I am excited to type ∀ and
```

……□□□□ `\-` □□□□□□□□□□□□ `>` □□ ` ` □□ `→` □ `≤` □□□□ `\<=` □ `≡` □□ `\==` □

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

□□□□□□□□□□……

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

□□□ Emacs □□ Unicode □□□□□□□□□□□□□□□□□□□ Unicode □□

□□ `agda-mode` □ Emacs □□□□□□□□□□□□□□ Unicode □□□□□□ □□□□□□□□□□ `agda-input-show-translations` □□□□

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

`agda-mode` □□□□□□□□□□

□□□□□□□ agda □□□□□□□ Unicode □□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□

Spacemacs

Spacemacs □□□□□□□ Emacs □□□□ Emacs □ Vim □□□□□□□□□□□□□□ `agda-mode` □□□□□□□□

Visual Studio Code

Visual Studio Code □□□□□□□□□□□□□□□□ Visual Studio □□□□ Agda □□□

Atom

Atom □□□ GitHub □□□□□□□□□□□ Atom □□□□□□ Agda □□□

□□□□□

PLFA □□ Pandoc Markdown □□□□□□ Agda □□□□ PLFA □□□□□□□□□ EPUB □□□□□□□□□□□□ UNIX □ ma-cOS □□□□ □□□□□□□□□□□□□□□□□□□□□□□□ Git □□□□□

□□□□□□□□

□□□□□□□□□ PLFA □□□□□□□□□ Stack□ PLFA □ Hakyll □□□□□□□□□□ Haskell □□□□□□ □□□□□□□□□□□□□□□□ Makefile□□□□□□ PLFA□□□□

```
make build
```

□□□□□□□□□ PLFA□□□□□□□□□□□□□□□□□□

```
make watch
```

Makefile □□□□□□□□□□□□□□□□□□□□□□□□

```
build                       # □□ PLFA
watch                       # □□□□□ PLFA□□□□□□□□□□□□
test                        # □□□□□□□□□□□□□□□ HTML □
test-epub                   # □□ EPUB □□□□□□□□ EPUB3 □□
clean                       # □□ PLFA □□
init                        # □□ Git □□□□□□□□□
update-contributors         # □ GitHub □□□□□□□□□□ contributors/
list                        # □□□□□□□□
```

□□□□□Makefile □□□□□□□□□□□□□□□□□□□□

```
legacy-versions             # □□□□□□ PLFA
setup-install-bundler       # □□ Ruby Bundler □‘legacy-versions’ □□□□
setup-install-htmlproofer   # □□ HTMLProofer □‘test’ □ Git □□□□□□□□
setup-check-fix-whitespace  # □□ fix-whitespace □□□□□ □Git □□□□□□□□
setup-check-epubcheck       # □□ epubcheck □□□□□ □EPUB □□□□□□
setup-check-gem             # □□ RubyGems □□□□□
setup-check-npm             # □□ Node □□□□□□□□□
setup-check-stack           # □□ Haskell □□ Stack □□□□□
```

□□□ EPUB □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Git □□□□

□□□□□□□□□ Git □□□

1. fix-whitespace [illegible]
2. [illegible]

[illegible] `make init` [illegible] Git [illegible] [illegible] fix-whitespace[illegible]

```
stack install fix-whitespace
```

[illegible] Stack [illegible] GHC[illegible] `--system-ghc` [illegible] `stack-*.yaml` [illegible] Agda [illegible]

Part I

[illegible]

Chapter 1

Naturals: □□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□ 7*10^22 □□□□□

□□ □□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□**Inductive Datatype**□

□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□**Type**□□□□ `ℕ` □□□□ `0`□`1`□`2`□`3` □□□ □□□□ `ℕ` □□□□**Value**□□□□□□□ `0 : ℕ`□`1 : ℕ`□`2 : ℕ`□`3 : ℕ` □□□□

□□ □□□□□□□**Inference Rules**□□□□□□□□□□□

```
--------
zero : ℕ

m : ℕ
---------
suc m : ℕ
```

□□□□ Agda □□□□□□□□

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

□□□ `ℕ` □□□□□□□□□□□□□□□**Datatype**□□□□□□□□ `zero` □□□□□ `suc` □□□□□□□ **Successor** □□□□□□□□□□□□□□□□□□□□**Constructor**□□

□□□□□□□□□□□□□□□□□

- [illegible]**Base Case**[illegible]`zero` [illegible]
- [illegible]**Inductive Case**[illegible] `m` [illegible] `suc m` [illegible]

[illegible]

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

[illegible] `zero` [illegible] `0` [illegible] `suc zero` [illegible] [illegible] `1` [illegible] `suc (suc zero)` [illegible] `suc 1` [illegible] [illegible] `2` [illegible] `3` [illegible]

[illegible] `seven` [illegible]

[illegible] `7` [illegible]

```
-- [illegible]
```

[illegible]

[illegible] [illegible]**Judgment**[illegible]**Hypothesis**[illegible] [illegible]**Conclusion**[illegible] [illegible] `zero` [illegible] [illegible] `m` [illegible] `suc m` [illegible]

Agda [illegible]

[illegible] Agda [illegible] `data` [illegible] [illegible]

```
ℕ : Set
```

[illegible] `ℕ` [illegible] `Set` [illegible] Agda [illegible] [illegible] `where` [illegible] [illegible] `data` [illegible]

```
zero : ℕ
suc  : ℕ → ℕ
```

[illegible] `zero` [illegible] `suc` [illegible]**Signature**[illegible] [illegible] `zero` [illegible] `suc` [illegible]

[illegible] `ℕ` [illegible] `→` [illegible] **Unicode**[illegible] Unicode [illegible] Emacs [illegible]

[illegible]

[illegible]

- [illegible]**Base Case**[illegible]`zero` [illegible]
- [illegible]**Inductive Case**[illegible] `m` [illegible] `suc m` [illegible]

[illegible]

[illegible]·[illegible]

[illegible]

[illegible]

```
-- [illegible]
```

[illegible] `zero` [illegible] `m` [illegible]`suc m` [illegible]

```
-- [illegible]
zero : ℕ
```

[illegible] `zero` [illegible] `zero` [illegible] `suc zero` [illegible]

```
-- [illegible]
zero : ℕ
suc zero : ℕ
```

[illegible] `zero` [illegible] `suc zero` [illegible] `suc zero` [illegible] `suc (suc zero)` [illegible] `suc zero` [illegible] `suc (suc zero)` [illegible]

```
-- [illegible]
zero : ℕ
suc zero : ℕ
suc (suc zero) : ℕ
```

[illegible]

```
-- [illegible]
zero : ℕ
suc zero : ℕ
suc (suc zero) : ℕ
suc (suc (suc zero)) : ℕ
```

[illegible] *n* [illegible] *n* [illegible] *n* [illegible] *n+1* [illegible] *n* [illegible] *n+1* [illegible]

[illegible]**Inductive**[illegible]

[illegible]

[illegible]

[illegible]

□□□ □□□□□□□□□□□·□□□□Richard Dedekind□□ 1888 □□□□□□□ *Was sind und was sollen die Zahlen?*“□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□·□□□□Giuseppe Peano□□□□□□□□□□□”*Arithmetices principia, nova methodo exposita*”□□□□□□□□□□□□□□□□□□□□□□□

□□□□

□ Agda □□□□□□ `--` □□□□□ `{-` □ `-}` □□□□□□□□□□□ □□□**Comment**□□**Pragma**□□□ `{-#` □ `#-}` □□□

```
{-# BUILTIN NATURAL ℕ #-}
```

□□□□□ Agda □□□□ `ℕ` □□□□□□□□□□□□□□□□□□□□ `zero` □□ □ `0` □□ `suc zero` □□□ `1` □□ `suc (suc zero)` □□□ `2` □□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□ `ℕ` □□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□ `zero` □□□□□□□□□□□□□□□□□□□□□□□□□□□ `suc` □□

□□□□□□□□□□□□□□□□□□□□□□ Haskell □□□□□□□□□□ □□□□□□□□□□□□□□□□□□□ `zero` □ `suc` □□□□□ *n* □□□□□ □ *n* □□□□□□□□□□□ Haskell □□□□□□□□□□□□□□□□□□□□ *n* □□□□□□□

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Agda □□□ □□□□□□□□**Equality**□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□**Scope**□□□ □□□□□□ `Eq` □□□□□□□□□□□□□□□□□□□□□□□□ `using` □□ `_≡_` □□□□□□□□□□ `refl` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `using` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `begin_` □ `_≡⟨⟩_` □ `_∎` □□□ □□□□□□□□□□□□□

Agda □□□□□□□□□**Term**□□□□□Infix□□□□□Mixfix□□□□□□□□□□□□□□ □□□ `_≡_` □ `_≡⟨⟩_` □□□□□□□□□□□□□□□□□□□ `begin_` □□□□ □□□□□□□□□□□□□□ `_∎` □□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `using` □□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□

□□ □□□□□□□□□□□□□□□□□□□□□**Recursion**□□

□□□ Agda □□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□ `_+_` □□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□ `ℕ → ℕ → ℕ` □□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `m + n` □ `_+_ m n` □□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□ `zero + n` □□ `n` □□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□ `(suc m) + n` □□ `suc (m + n)` □□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□**Pattern Matching**□□

□□□□□ `zero` □□ `0` □□ `suc m` □□ `1 + m` □□□□□□□□□□ □□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
(m + n) + p  ≡  m + (n + p)
```

□□□□□□□□□□ `m` □□ `1` □ `n` □□ `m` □ `p` □□ `n` □□□□ □□□□□□□□□□□□□□ `=` □□□□□□ `≡` □□□□□□□□□□□□□

□□□□□□□□□**Recursive**□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□**Well founded**□□

□□□□□□□□□□□□

```
_ : 2 + 3 ≡ 5
_ =
  begin
    2 + 3
  ≡⟨⟩ -- □□□
    (suc (suc zero)) + (suc (suc (suc zero)))
  ≡⟨⟩ -- □□□□
    suc ((suc zero) + (suc (suc (suc zero))))
  ≡⟨⟩ -- □□□□
    suc (suc (zero + (suc (suc (suc zero)))))
  ≡⟨⟩ -- □□□□
    suc (suc (suc (suc (suc zero))))
  ≡⟨⟩ -- □□□
    5
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□ `m = 1` □ `n = 3` □□□□□□□□□□□□ `m = 0` □ `n = 3` □□□□□□□□□□□□ `n = 3` □□□□□□□□

□□□□□□□□□□□□□□□□□□□□□ `:` □□□□□□□□□□□□□ □□□□□□Binding□□□□□ `=` □□□□□□□□□□□□□□□□□□□ □□□□□□□□ `_` □□□□□□□□□□□□□□□□□□□□□□□□ `_` □□ □□□□□□□□□□□□□□□□□

□□□□□□□ `2 + 3 ≡ 5` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□**Evidence**□□□□□□□□ `begin` □□□□ `∎` □□□ `∎` □□□□qed□□□□□□□□□tombstone□□□□□□□□□□□□□□□□□□□□□□ □□□□□ `≡⟨⟩` □□□□□□□□

□□□ Agda □□□□□

```
_ : 2 + 3 ≡ 5
_ = refl
```

Agda □□□□□□□ `2 + 3` □□□□□□□□□□□□□□□□□ `5` □□□□□□□□□□□ □□□□□□Binary Relation□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□**Reflexivity**□□□ Agda □□□□□□□□□□□□□□□□□□□□□□ `refl` □

□□□□□□□Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□

□□□□□ `2 + 3 ≡ 5` □□□□□□□□□□□□□□□□ `refl` □□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□ `2 + 3 ≡ 5` □□□□□□□□□□□□□□□□□□——□□□□□□□□□□□□□□□——□□□□□ Agda □□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□

□□□ □□

□□ `+-example` □□□□

□□ `3 + 4` □□□□□□□□□□□□□□□□□□□ `+` □□□□□□

```
-- □□□□□□□□□□□□
```

□□

□□□

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

□□ `m * n` □□□□□□ `m` □ `n` □□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□ □□

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

□□□□□□□□□□□□ `m` □□ `1` □ `n` □□ `m` □ `p` □□ `n` □□□□ □□□□□□□□□□□□□□□□□ `1 * n ≡ n` □□□□□□□□□□□□□□□□□□□□□

□□□

□□□□□□□□□□□□□□

```
_ =
  begin
    2 * 3
  ≡⟨⟩ -- □□□□
    3 + (1 * 3)
  ≡⟨⟩ -- □□□□
    3 + (3 + (0 * 3))
  ≡⟨⟩ -- □□□□
    3 + (3 + 0)
  ≡⟨⟩ -- □□
    6
  ∎
```

□□□□ `m = 1` □ `n = 3` □□□□□□□□□□□□ `m = 0` □ `n = 3` □□□□□□□□□□□□□□ `n = 3` □□□□□□□□□□□□□□□□ `_ : 2 * 3 ≡ 6` □□□□□□□□□□□□□□□□□□□□

□□ `*-example` □□□□

□□ `3 * 4` □□□□□□□□□□□□□□□ `*` □□□□□ □□□□□ `+` □□□□□□□□

```
-- □□□□□□□□
```

□□ `_^_` □□□□

□□□□□□□□□□□□□□

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

□□ `3 ^ 4` □□□□ `81` □

```
-- □□□□□□□□
```

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□**Monus**□□ **minus** □□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□

- □□□□□□□□
  - □□□□ `zero` □□□□□□□□□

  - □□□□ `suc n` □□□□□□□□□
    * □□□□ `zero` □□□□□□□□□
    * □□□□ `suc m` □□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□ `∸-example₁` □ `∸-example₂` □□□□ {name=monus-examples}

□□ `5 ∸ 3` □ `3 ∸ 5` □□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□□

□□□□□□□□□□**Precedence**□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□ `suc m + n` □□□ `(suc m) + n` □□□□□□□□□□□□□□ □□□□□□□□□□□□□□ `n + m * n` □□□ `n + (m * n)` □□□□ □□□□□□□□□□□□□□□□□ `m + n + p` □□□ `(m + n) + p`□

□ Agda □□□□□□□□□□□□□□□□□□□□□

```
infixl 6 _+_ _∸_
infixl 7 _*_
```

□□□□□□□ `_+_` □ `_∸_` □□□□□ 6□□□□ `_*_` □□□□ □ 7□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `infixl` □□□□□□□□□□□□□□□□□□□□□□ `infixr` □□□ □□□□□□□□□□□□□□ `infix` □□□□□□□□□□□□□□□

[illegible]

[illegible] [illegible]**Currying**[illegible]

[illegible] Haskell [illegible] ML [illegible]Agda [illegible] [illegible]

[illegible]

`ℕ → ℕ → ℕ` [illegible] `ℕ → (ℕ → ℕ)`

[illegible]

`_+_ 2 3` [illegible] `(_+_ 2) 3`[illegible]

`_+_ 2` [illegible]

[illegible]·[illegible]Haskell Curry[illegible] Haskell [illegible] [illegible] 19 [illegible] 30 [illegible] 20 [illegible] Moses Schönfinkel [illegible] Schönfinkel [illegible]Curry[illegible] [illegible]·[illegible] [illegible]Gottlob Frege[illegible] 1879 [illegible] **“Begriffsschrift”**[illegible]

[illegible]

[illegible]

[illegible] [illegible]

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

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

[illegible] `n : ℕ` [illegible] [illegible] `n` [illegible] `n`[illegible] `m` [illegible] `n` [illegible] `p` [illegible] `suc m` [illegible] `n` [illegible] `suc p`[illegible]

[illegible]

```
-- [illegible]
```

[illegible] [illegible] `n` [illegible] `zero + n = n`[illegible] `m + n = p`[illegible] [illegible] `suc m + n = suc p`[illegible]

```
-- [illegible] 0 [illegible]
0 + 0 = 0     0 + 1 = 1    0 + 2 = 2     ...
```

[illegible] [illegible]

```
-- [illegible] 0[illegible]1 [illegible]
0 + 0 = 0     0 + 1 = 1     0 + 2 = 2     0 + 3 = 3     ...
1 + 0 = 1     1 + 1 = 2     1 + 2 = 3     1 + 3 = 4     ...
```

[illegible]

```
-- [illegible] 0[illegible]1[illegible]2 [illegible]
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     ...
```

[illegible]

```
-- [illegible] 0[illegible]1[illegible]2[illegible]3 [illegible]
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     ...
```

[illegible] *m* [illegible] *m* [illegible]

[illegible]

[illegible]

[illegible] [illegible]

[illegible] [illegible]

```
-- [illegible]
```

[illegible]

```
-- [illegible]
0 : ℕ
```

[illegible]

```
-- [illegible]
0 : ℕ
1 : ℕ    0 + 0 = 0
```

[illegible] [illegible]

```
-- [illegible]
0 : ℕ
1 : ℕ    0 + 0 = 0
2 : ℕ    0 + 1 = 1   1 + 0 = 1
```

[illegible]

```
-- [illegible]
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
```

[illegible] *n* [illegible] *n* [illegible] *n × (n-1) / 2* [illegible] *n* [illegible] *n* [illegible] *n+1* [illegible]

[illegible]

Agda [illegible] Emacs [illegible] [illegible]

[illegible]

```
_+_ : ℕ → ℕ → ℕ
m + n = ?
```

[illegible] Agda [illegible] `C-c C-l` [illegible] Control [illegible] `c` [illegible] `l` [illegible]`l` [illegible] **l**oad[illegible]

```
_+_ : ℕ → ℕ → ℕ
m + n = { }0
```

[illegible]**Hole**[illegible]0 [illegible] [illegible]Emacs [illegible]

```
?0 : ℕ
```

[illegible] 0 [illegible] `ℕ` [illegible] `C-c C-f` [illegible]`f` [illegible] **f**orward[illegible]

[illegible] [illegible] 0 [illegible] `C-c C-c`[illegible]`c` [illegible] **c**ase[illegible]

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

[illegible]

[illegible] `m` [illegible] `m` [illegible]

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

[illegible]

```
?0 : ℕ
?1 : ℕ
```

[illegible] 0 [illegible] `C-c C-,` [illegible] [illegible]

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

[illegible] `n` [illegible] `C-c C-`[illegible] [illegible]

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

[illegible] 1 [illegible] `C-c C-,` [illegible] [illegible]

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

□□□□□□□□□ `C-c C-r` □ `r` □□□□□ **r**efine□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□“□□□□ `zero` □ `suc` □□□□□□□□□□”□

□□□ `suc ?` □□□□□ `C-c C-□□` □□□□□□□□□□

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

□□□□□□□□□ `C-c C-,` □□□□□□□□□□□□

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

□□□□□ `m + n` □□□□□□ `C-c C-□□` □□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `C-c C-c` □□□□□□□□□□

□□□□□

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

□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□ Haskell □□□□ □□ `m` □ `n` □□□ `zero` □ `suc` □□□□□□□□□□□ `m` □□□□ □□ Haskell □□□□□□□□□□□□□□ `m` □ `n` □□□□□□□□□□□ □□□□□□ `m` □ `n` □□□ `zero` □ `suc` □□□□□□□□□□□ `m` □ `n` □ □□□□□ Haskell □□□□□□□□□□□□□□ `m` □ `n` □□□□□□□□□

□□ `Bin` □□□□

□□□

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

□□□□□□□□

```
1011
```

□□□□□□□□□□□

```
⟨⟩ I O I I
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□ `001011` □□□□□

```
⟨⟩ O O I O I I
```

□□□□□□□□

```
inc : Bin → Bin
```

□□□□□□□□□□□□□□□□□□□□□`1100` □□□□□□□□□□□□□

```
inc (⟨⟩ I O I I) ≡ ⟨⟩ I I O O
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□

```
to   : ℕ → Bin
from : Bin → ℕ
```

□□□□□□□□□□□□□□□□□□□□□□□ `⟨⟩ O` □□□□ □□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `Data.Nat` □□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□Agda □□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□ `NATURAL` □□□□□□□□□ `ℕ` □□□□□□□□□□ □□□□□ `Data.Nat.ℕ` □□□□□□□□□□□□□□□□□□□□□ □□□□□□□ `2` □□□□□ `ℕ` □□□□□□ `Data.Nat.ℕ` □ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Agda □□□□□□

Unicode

□□□□□□□□□ Unicode □□□

```
ℕ  U+2115  □□□□□ N (\bN)
→  U+2192  □□□ (\to, \r, \->)
∸  U+2238  □□ (\.-)
≡  U+2261  □□□ (\==)
⟨  U+27E8  □□□□□□ (\<)
⟩  U+27E9  □□□□□□ (\>)
∎  U+220E  □□ (\qed)
```

□□□□□□□□□ Unicode □□□□ `ℕ` □□□□□ Unicode □□□□ `U+2115` □□ □□□□□□□ `□□□□□ N` □□□□□□□ Emacs □□□□□□□□□□□□□ `\bN` □□

[illegible] `\r` [illegible] `\r` [illegible] `\l` [illegible]

[illegible]

```
C-b  [illegible]
C-f  [illegible]
C-p  [illegible]
C-n  [illegible]
```

`C-b` [illegible] Control + b[illegible]

[illegible] `agda-input-show-translations` [illegible]

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

[illegible] `agda-mode` [illegible] M-x [illegible] `ESC` [illegible] `x` [illegible]

[illegible] agda [illegible] Unicode [illegible] `quail-show-key` [illegible]

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

[illegible] `∸` [illegible] `M-x quail-show-key` [illegible] `\.-` [illegible]

Chapter 2

Induction: □□□□

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

> □□□□□□□□□□□□□□□□ ……□□□□□□□□□□□□□□□□□□□ —— Herbert Wilf

□□ □□□□□□□□□□□□□□**Inductive Datatype**□□□□□□□□□**Induction**□□□□□□

□□

□□□ `cong`□`sym` □ `_≡⟨_⟩_` □□□□□□□□□□

```
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; _+_; _*_; _∸_)
```

□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- □□□**Identity**□□□□□□□□ `n` □□ `0 + n ≡ n` □□ `+` □□□□ `0`□ □ `n + 0 ≡ n` □□ `+` □□□□ `0` □□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□**Unit**□□
- □□□□**Associativity**□□□□□□□□□□□□□□□□□□□□□□ `+` □□□□□□□ □□□□□□□ `m`□`n` □ `p` □□ `(m + n) + p ≡ m + (n + p)`□
- □□□□**Commutativity**□□□□□□□□□□□□□□□□□□□□□□ `+` □□□□□□□ □□□□□□□ `m` □ `n` □□ `m + n ≡ n + m`□
- □□□□**Distributivity**□□□□□□□□ `m`□`n` □ `p` □□ `(m + n) * p ≡ (m * p) + (n * p)` □□□□□ `*` □□□□ `+` □□□□□□□ □□□□□□ `m`□`n` □ `p` □□ `m * (p + q) ≡ (m * p) + (m * q)` □□□□□□□□□□

□□□□□□ `0` □□□□□□□ `1` □□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□

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.

[illegible] [illegible]

[illegible]Operator[illegible]

[illegible] [illegible] [illegible] [illegible]

[illegible] `operators` [illegible]

[illegible] [illegible]

```
-- [illegible]
```

[illegible] [illegible]

```
-- [illegible]
```

[illegible]

[illegible]

```
(m + n) + p ≡ m + (n + p)
```

[illegible] `m`[illegible]`n` [illegible] `p` [illegible]

[illegible]

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

[illegible] [illegible] `12` [illegible]

[illegible] `7 + 5` [illegible] `3 + 9` [illegible] [illegible] [illegible]

[illegible]**Proof by Induction**[illegible] [illegible]

[illegible]

[illegible] `zero` [illegible] [illegible] `m` [illegible] `suc m` [illegible]

[illegible] `zero` [illegible] `m` [illegible]**Induction Hypothesis**[illegible] `suc m` [illegible]

[illegible] `m` [illegible]Property[illegible] `P m`[illegible]

```
------
P zero

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

[illegible] `P` [illegible] `zero` [illegible]`P` [illegible] `m` [illegible] `P` [illegible] `suc m` [illegible]

[illegible]

```
-- [illegible]
```

[illegible] `P zero` [illegible] [illegible]`P m` [illegible]`P (suc m)` [illegible] [illegible]

```
-- [illegible]
P zero
```

[illegible] [illegible] `P zero` [illegible] `P zero` [illegible] `P (suc zero)` [illegible]

```
-- [illegible]
P zero
P (suc zero)
```

[illegible] `P zero` [illegible] `P (suc zero)` [illegible] [illegible] `P (suc zero)` [illegible] `P (suc (suc zero))` [illegible] [illegible]

```
-- [illegible]
P zero
P (suc zero)
P (suc (suc zero))
```

[illegible]

```
-- [illegible]
P zero
P (suc zero)
P (suc (suc zero))
P (suc (suc (suc zero)))
```

[illegible] *n* [illegible] *n* [illegible] [illegible] `P n` [illegible] *n+1* [illegible]

□□□□□□□□□

□□□□□□□□□□□ `P m` □□□□□□

```
(m + n) + p ≡ m + (n + p)
```

□□□ `n` □ `p` □□□□□□□□□□□□□□□□□□□□□ `m` □□□□□□□□□□□□ `n` □ `p` □□□□□□□□□□□□□□□

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

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□

```
+-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)
  ∎
```

□□□□□□□□ `+-assoc` □□ Agda □□□□□□□□□□□ `@.(){};_` □□□□□□□□□□□

□□□□□□□□□□□□□□□Signature□□□□□□□□□□□ `+-assoc` □□□□□□□□□□Evidence□□

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

□ A □□□□□□□□□for all□□□□□□□□□□□□□□□□ `m`□`n` □ `p` □□□ `(m + n) + p ≡ m + (n + p)` □□□□□□□□□□□□□□□□□□□□□ □□□□□□ `m`□`n` □ `p` □□□□□□□□□□□□□□

□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□

```
n + p ≡ n + p
```

□□□ □□□□□□□□□□□□□□□□□□ `n + p` □□□□□□□□□□□□□

□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□

```
(m + n) + p ≡ m + (n + p)
```

□□□□□ `suc` □□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□ `_≡⟨_⟩_` □ □□□□□□□□□□□□□□□□□□□□□

```
⟨ cong suc (+-assoc m n p) ⟩
```

□□□□□□□□ `+-assoc m n p` □□□□□□□□□□ `cong suc` □□□□□□□□□□ `suc` □□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□**Congruence**□□ □ `e` □ `x ≡ y` □□□□□□□□□□□ `f` □ `cong f e` □□ `f x ≡ f y` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□ `+-assoc m n p` □□□□ □□□□□□□□□□well-founded□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□`assoc (suc m) n p` □□ `assoc m n p` □□□□ □□□□□□□□□□□□□□□□ Agda □□□□□□□□□□

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `m` □□□□ `2` □□□□□□

```
+-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)
      ∎
```

□□□□□

□□□□□□□□□□□□□□ `∀` □□□□□□□□□ `m`□`n` □ `p` □□□□ □□□ `∀` □□□□□□（Universal Quantifier）□□□□□□ Quanti-fiers □□□□□□□□□

□□□□□□□□□□□□□□□□

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

□

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

□□□□□□ `ℕ → ℕ → ℕ` □□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□（Dependent Function）□□

□□□□□□□

□□□□□□□□□□□□□□□□□□（**Commutativity**）□□□□□□□□□□□□□□□

```
m + n ≡ n + m
```

□□□□□□□□□□□□□□□□□□（Lemma）□□

□□□□□

□□□□□□□□□□□□□□□□□□□□

```
zero + n ≡ n
```

□□□□□□□□□□□□□□□□□□□□□

```
m + zero ≡ m
```

□□□□□□□□□□

```
+-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
  ∎
```

□□□□□□□□□□□□□ `+-identityʳ` □□□□□□□□□□□

```
∀ (m : ℕ) → m + zero ≡ m
```

□□□□□□□□□□□□□□□□□□□□□□□□□□ `m`□□□□□ □□□□□□□□□□□□□□□ `m` □□□□□□□□

□□□□□□□□□□□□□□

```
zero + zero ≡ zero
```

□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□

```
m + zero ≡ m
```

□□□□□□ `suc` □□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□

```
⟨ cong suc (+-identityʳ m) ⟩
```

□□□□□□□□□ `+-identityʳ m` □□□□□□□□□□□ `cong suc` □□□□□□□□□□□ `suc` □□□□□□□□□□□□□□□□□

□□□□

□□□□□□□□□□□□□□□□ `suc` □□□□□□

```
suc m + n ≡ suc (m + n)
```

□□□□□□□□□□□□□□□□ `suc` □□□□□□□

```
m + suc n ≡ suc (m + n)
```

□□□□□□□□□□□□□

```
+-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)
  ∎
```

□□□□□□□□□□□□□ `+-suc` □□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `m` □ `n`□ □□□□□□□□□□□□□□□□□□ `m` □□□□□□□□

□□□□□□□□□□□□□□

```
zero + suc n ≡ suc (zero + n)
```

□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□

```
m + suc n ≡ suc (m + n)
```

□□□□□□ `suc` □□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□

```
⟨ cong suc (+-suc m n) ⟩
```

□□□□□□□□□ `+-suc m n` □□□□□□□□□□□ `cong suc` □□□□□□□□□□□ `suc` □□□□□□□□□□□□□□□□□

□□

□□□□□□□□□□□□□□□□□□

```
+-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
  ∎
```

□□□□□□□□□□□□□ `+-comm` □□□□□□□□□□□

```
∀ (m n : ℕ) → m + n ≡ n + m
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `m` □ `n` □ □□□□□□□□□□□□□□□□□□ `n` □□□□□□□□□□□□□ `m` □□

□□□□□□□□□□□□□□

```
m + zero ≡ zero + m
```

□□□□□□□□□□□□□□□□□□

```
m + zero ≡ m
```

□□□□□□□□□ `⟨ +-identityʳ m ⟩` □□□□□□□□□□□□

□□□□□□□□□□□□□□

```
m + suc n ≡ suc n + m
```

□□□□□□□□□□□□□□□□□□

```
m + suc n ≡ suc (n + m)
```

□□□□□□□□□□□□□□□□□

```
m + suc n ≡ suc (m + n)
```

□□□□□□□□ `⟨ +-suc m n ⟩` □□□□□□□□□

```
suc (m + n) ≡ suc (n + m)
```

□□□□□□□□□□□□ `⟨ cong suc (+-comm m n) ⟩` □□□□□□

Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□

□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□

```
+-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
  ∎
```

□□□

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

□□□□□□ `sym` □□□□□□□□□□□□ `+-assoc n p q` □□□□□□□□□□□□□

```
(n + p) + q ≡ n + (p + q)
```

□□□□□□□□□□□□□□□□□ `sym (+-assoc m n p)`□

```
n + (p + q) ≡ (n + p) + q
```

□□□□□□ `e` □□□ `x ≡ y` □□□□□□ `sym e` □□□□ `y ≡ x` □□□□

□□□Agda □□ Richard Bird □□□□□□**Section**□□□□□□□□□□ `y` □□□ `x + y` □□□□□ `(x +_)` □□□□□□□□□ `cong (m +_)` □□□□□□□□□□□

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

□□□□□□□□□□ `y` □□□ `y + x` □□□□□ `(_+ x)`□ □□□□□□□□□□□□□□

□□□□□□□

□□□ □□□□□□□□□□□□□□□□□

```
  -- □□□□□□□□□□□□□□
```

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

```
-- □□□□□□□□□□ 0 □□□□□
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□□ 0 □ 1 □□□□□
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
(1 + 0) + 0 ≡ 1 + (0 + 0)   ...   (1 + 4) + 5 ≡ 1 + (4 + 5)   ...
```

□□□□□□□□□□

```
-- □□□□□□□□□□ 0□1 □ 2 □□□□□
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
(1 + 0) + 0 ≡ 1 + (0 + 0)   ...   (1 + 4) + 5 ≡ 1 + (4 + 5)   ...
(2 + 0) + 0 ≡ 2 + (0 + 0)   ...   (2 + 4) + 5 ≡ 2 + (4 + 5)   ...
```

□□□□□□□□□□□

```
-- □□□□□□□□□□ 0□1□2 □ 3 □□□□□
(0 + 0) + 0 ≡ 0 + (0 + 0)   ...   (0 + 4) + 5 ≡ 0 + (4 + 5)   ...
(1 + 0) + 0 ≡ 1 + (0 + 0)   ...   (1 + 4) + 5 ≡ 1 + (4 + 5)   ...
(2 + 0) + 0 ≡ 2 + (0 + 0)   ...   (2 + 4) + 5 ≡ 2 + (4 + 5)   ...
(3 + 0) + 0 ≡ 3 + (0 + 0)   ...   (3 + 4) + 5 ≡ 3 + (4 + 5)   ...
```

□□□□□□□□□□□□ *m* □□□□□□□□□□□□□□ *m* □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `finite-|-assoc` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□
```

□□□□□□□□

□□□□□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□ `rewrite` □□□□ □□□□□□

```
+-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
```

□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□

```
n + p ≡ n + p
```

[illegible] `refl` [illegible]

[illegible]

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

[illegible]

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

[illegible] `refl` [illegible] [illegible] `rewrite` [illegible] [illegible] `cong`.

[illegible]

[illegible] `rewrite` [illegible]

```
+-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
```

[illegible] [illegible]

[illegible]

[illegible] Emacs [illegible] Agda [illegible] [illegible]

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

[illegible] Agda [illegible] `C-c C-l` [illegible] Ctrl-c [illegible] Ctrl-l[illegible]

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

[illegible]**Hole**[illegible]0 [illegible] Emacs [illegible]

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

[illegible] 0 [illegible]

[illegible] `m` [illegible] `C-c C-c` [illegible]

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

□□ `m` □□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□ 0 □□□□□ `C-c C-,` □□□□□□□□

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

□□□□□□□□□0 □□□□□□□□□□□□□□□□□ `p` □ `n` □□□□□□□□ □□□□□□□□□□□□□□□□□□□□□ `C-c C-r` □□□□□□□ `C-c C-l` □□□□□□□□□□□□ 0□

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

□□□□ 0 □□□□□ `C-c C-,` □□□□□□□□

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

□□□ □□□□□□□□□□□□

```
+-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
```

□□□□□□□□□□ `C-c C-,` □□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□ `C-c C-r` □□□□□□□□□□

```
+-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
```

□□□ `+-swap` □□□□

□□□□□□□□□□□ `m`□`n` □ `p`□

```
m + (n + p) ≡ n + (m + p)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□ `*-distrib-+` □□□□

□□□□□□□□□□□□□□□□□□□□□□□ `m`□`n` □ `p`□

```
(m + n) * p ≡ m * p + n * p
```

□□□

```
-- □□□□□□□□□
```

□□ `*-assoc` □□□□

□□□□□□□□□□□□□□□□□□□□ `m`□`n` □ `p`□

```
(m * n) * p ≡ m * (n * p)
```

□□□

```
-- □□□□□□□□□
```

□□ `*-comm` □□□□

□□□□□□□□□□□□□□□□□□□□ `m` □ `n`□

```
m * n ≡ n * m
```

□□□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□ `0∸n≡0` □□□□

□□□□□□□□□□□ `n`□

```
zero ∸ n ≡ zero
```

□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□ `∸-|-assoc` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□ `m`□`n` □ `p`□

```
m ∸ n ∸ p ≡ m ∸ (n + p)
```

□□□

```
-- □□□□□□□□□
```

□□ `+*^` □□□□

□□□□□□□□

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

□□□□ `m`□`n` □ `p` □□□

□□ `Bin-laws` □□□□

□□□□ Bin □□□□□□□□□□□□□□□□□□□ `Bin` □□□□□□□□□□□

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

□□□□□□□□□ `n` □□□□□□`b` □□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□

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

Unicode

□□□□□□□□□ Unicode□

```
∀  U+2200  □□□□□ (\forall, \all)
ʳ  U+02B3  □□□□□□□□□ r (\^r)
′  U+2032  □□ (\')
″  U+2033  □□□ (\')
‴  U+2034  □□□ (\')
⁗  U+2057  □□□ (\')
```

□ `\r` □□□□□ `\^r` □□□□□□□□□□□□□□□□□□□□□ `r`□ □□ `\'` □□□□□□□□`′ ″ ‴ ⁗`□□

Chapter 3

Relations: □□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□**Relation**□□□□□□□□

□□

```
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)
```

□□□□

□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
z≤n --------
    zero ≤ n

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

□□□ Agda □□□

```
data _≤_ : ℕ → ℕ → Set where

  z≤n : ∀ {n : ℕ}
```

```
    --------
  → zero ≤ n

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

□□□□ `z≤n` □ `s≤s` □□□□□□□□□□□□□□ `zero ≤ n` □ `m ≤ n` □ `suc m ≤ suc n` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□**Indexed datatype**□□□□□□ `m` □ `n` □□□□□□□□□ `m ≤ n` □□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□

- □□□□: □□□□□□□□□ `n` □□□ `zero ≤ n` □□□
- □□□□□□□□□□□□□□□□ `m` □ `n` □□□□□ `m ≤ n` □□□ □□□□ `suc m ≤ suc n` □□□

□□□□□□□□□□□□□□□□□□□□□

- □□□□: □□□□□□□□□ `n` □□□□ `z≤n` □□□ `zero ≤ n` □□□□□□□
- □□□□□□□□□□□□□□□□ `m` □ `n` □□□□ `s≤s` □ `m ≤ n` □□□□□ □□□ `suc m ≤ suc n` □□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□ `2 ≤ 4` □□□□

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

□□□□□□□ Agda □□□

```
_ : 2 ≤ 4
_ = s≤s (s≤s z≤n)
```

□□□□

□□□ `∀` □ □□□□□□□□□□□□□□□□□ `∀` □□□

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

□□□□□□□□□□□□□□□□□□□□ `{ }` □□□□□□□□ `( )` □ □□□□□□□□□□□□□□□**Implicit**□□□□□□□□□□□□□□□□□□□□□Agda □□□□□□□□□□□□**Infer**□□□□□□□□□□□□□□□□□ `m + n ≡ n + m` □□□□□□□□□□ `+-comm m n` □□ `zero ≤ n` □□□□□□□□□□□ `n` □□□□□□□ `m≤n` □ `m ≤ n` □□□□□ □□□□□□□□ `s≤s m≤n` □□ `suc m ≤ suc n` □□□□□□□□□□ `m` □ `n` □

□□ `2 ≤ 4` □ Agda □□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□ `_` □□ Agda □□□□□□□□□□□□□□□□□□□ □□□□□□□□□□ `+-identityʳ` □□□□□□□□□□□□□□

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

□□□ `_` □□ Agda □□□□□□□□□□□□□□□□ `m` □□□□□□□□□□□□□□□□ Agda □□□□□□□□ □□ Agda □□□□□□□□□□□□□□□□

□□□

□□□□□□□□□□□□□

```
infix 4 _≤_
```

□□□ `_≤_` □□□□□□□□ 4□□□□□□□□□□□ 6 □ `_+_` □□□□□□□□□ `1 + 2 ≤ 3` □□□□□ `(1 + 2) ≤ 3`□□□□ `infix` □□□□□□□□□□□□□□ □□□□□□□□□□ `1 ≤ 2 ≤ 3` □□□ `(1 ≤ 2) ≤ 3` □□ `1 ≤ (2 ≤ 3)` □□□□□□

□□□□

□□ □□□□□□ Decidable □□□□□□□□□□□

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `m ≤ n` □□ `suc m ≤ suc n` □□□□□□□ `suc m` □ `m` □□ □□□□□□□□□□□□□□`suc n` □□ `n` □□□□□□□□□ □□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□ `m` □ `n` □ `suc m ≤ suc n`□ □□□□□□□□□invert□□□□□□

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

□□□ `m≤n` □□□□□□□□□□□□□□ `m ≤ n` □□□□□□□□□□□ □□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□ `z≤n` □□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□

□□□□□□□□□□□□□□□□□□□□□

- □□□**Reflexive**□□□□□□□ `n` □□□ `n ≤ n` □□□
- □□□**Transitive**□□□□□□□ `m` □ `n` □ `p` □□□ `m ≤ n` □ `n ≤ p` □□□□□ `m ≤ p` □□□□
- □□□□**Anti-symmetric**□□□□□□□ `m` □ `n` □□□ `m ≤ n` □ `n ≤ m` □□□□□□□ `m ≡ n` □□□
- □□□**Total**□□□□□□□ `m` □ `n` □ `m ≤ n` □□ `n ≤ m` □□□

`_≤_` □□□□□□□□□□□

□□□□□□□□□□□□□□□□□□

- □□□**Preorder**□□□□□□□□□□□□□□
- □□□**Partial Order**□□□□□□□□□□□□
- □□□**Total Order**□□□□□□□□□□□

□□□ □□□□□□□□□□□□□□□□□□□□□

□□ □□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `orderings` □□□□

□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□ `n` □□□ `n ≤ n` □□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□ `n` □□□□□□□□□□□□□`zero ≤ zero` □ `z≤n` □□□□□□□□□□ □□□□ `≤-refl {n}` □□□□□□ `n ≤ n` □□□□□□□□□□□ `s≤s` □□□□□□ `suc n ≤ suc n` □□□□

□ Emacs □□□□□□□□□□□□□□□□□□□□□□□□□□□□ `C-c C-c` □ `C-c C-,` □ `C-c C-r` □□□

□□□

□□□□□□□□□□□□□□□□□□□□□□ `m` □ `n` □□□ `m ≤ n` □ `n ≤ p` □□□□□ `m ≤ p` □□□□□□`m`□`n` □ `p` □□□□□□

```
≤-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)
```

□□□□□ `m ≤ n` □□□□**Evidence**□□□□□□□□□□□□□□□□□□□□□□ `z≤n` □□□□ □□□□□□□ `z≤n` □□□□□□□□`n ≤ p` □□□□□□□□□□□□ `_` □□□□□□ □□□□□□□

□□□□□□□□□□□□□□□ `s≤s m≤n` □□□□□□□□□□□□ `s≤s n≤p` □□□□ □□□□□□ `suc m ≤ suc n` □ `suc n ≤ suc p` □□□ `suc m ≤ suc p` □ □□□□□□ `≤-trans m≤n n≤p` □□□□□ `m ≤ p` □□□□□□□ `s≤s` □□□□

`≤-trans (s≤s m≤n) z≤n` □□□□□□□□□□□□□□□□□□□□□□□□□ `suc n` □ □□□□□□□□□□□□□□□□ `zero` □Agda □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□

□□□□□□□□□□□□□□□□

```
≤-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)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□ `m ≤ n` □□□□□□□□□□□□□□□□□□□□□□□□□ □□□□ `m` □□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ Emacs □□□□□□□□□□□□□□□□□□□□□□□□□□□□ `C-c C-c` □ `C-c C-,` □ `C-c C-r` □□□

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□ `m` □ `n` □□□ `m ≤ n` □ `n ≤ m` □□□□□□□ `m ≡ n` □□□

```
≤-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)
```

□□□□□□□□ `m ≤ n` □ `n ≤ m` □□□□□□□□□

□□□□□□□□□□□□□□□□□ `z≤n` □□□□□□□□□□□□ `zero ≤ zero` □ `zero ≤ zero`□ □□ `zero ≡ zero` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□ `s≤s m≤n` □□□□□□□□□□□□□ `s≤s n≤m` □□□□□□□□□□□□ `suc m ≤ suc n` □ `suc n ≤ suc m` □□□ `suc m ≡ suc n` □□□□□□ `≤-antisym m≤n n≤m` □□□□□ `m ≡ n` □□□□□□□□□□□□□□□□□□□□

□□ `≤-antisym-cases` □□□□

□□□□□□□□□□□□□□□□ `z≤n` □□□□□□□□ `s≤s` □□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□ `m` □ `n` □ `m ≤ n` □□ `n ≤ m` □□□ □ `m` □ `n` □□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□

```
data Total (m n : ℕ) : Set where

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

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

`Total m n` □□□□□□□□□□□□□ `forward m≤n` □□ `flipped n≤m` □□□ `m≤n` □ `n≤m` □□□ `m ≤ n` □ `n ≤ m` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Disjunction□□□□□ □□□□ Connectives □□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□ `m` □ `n` □□□□□□□□□□□□□□□□□□□□□

```
data Total′ : ℕ → ℕ → Set where

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

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

□□ `zero ≤ n` □ `suc m ≤ suc n` □□□□□□□□□□□□□□□□□□□□□□□□□□ □□□ `Total m n` □□□□□□□□□□□□□□□□□□□□□□□□□□□□ Agda □ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□

```
≤-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)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□

- □□□□□□□□□□□□□□□ `zero` □□□□□□□ `n` □□□ forward □□□□□□□□□□ `z≤n` □□ `zero ≤ n` □□□□
- □□□□□□□□□□□□□□□ `suc m` □□□□□□□ `zero` □□□ flipped □□□□□□□□□ `z≤n` □□ `zero ≤ suc m` □□□□
- □□□□□□□□□□□□□ `suc m` □□□□□□□ `suc n` □□□□□□□ `≤-total m n` □□□□□□□□□
  - □□□□□ forward □□□□□□ `m≤n` □□ `m ≤ n` □□□□□□□□□□□□ `s≤s m≤n` □□ `suc m ≤ suc n` □□□ forward □□□□□
  - □□□□□ flipped □□□□□□ `n≤m` □□ `n ≤ m` □□□□□□□□□□□□ `s≤s n≤m` □□ `suc n ≤ suc m` □□□ flipped □□□□□

□□□□□□□□ Agda □□□ `with` □□□ `with` □□□□□□□□□□□□□□□□□ □□□□□□□ `...` □□□□□□□ `|` □□□□□□□□□□□□□□□□□□□□□□□□

□□ `with` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
≤-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)
```

□□□□□□□□□ Agda □□□ `where` □□□ `where` □□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□ `m` □ `n` □□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□ `helper` □□□□□□□□□□□□□□

□□ forward □□□ □□□□□□□□□ flipped □□□□□□

```
≤-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)
```

□□

□□□

□□□ □□□□**Monotonic**□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□

```
+-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)
```

□□□□□□□□□□□

- □□□□□□□□ `zero` □□ `zero + p ≤ zero + q` □□□□ `p ≤ q`□ □□□ `p≤q` □□
- □□□□□□□□ `suc n` □□ `suc n + p ≤ suc n + q` □□□□ `suc (n + p) ≤ suc (n + q)` □□□□ `+-monoʳ-≤ n p q p≤q` □□□ `n + p ≤ n + q` □□□□□□□ `s≤s` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
+-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
```

□ `+-comm m p` □ `+-comm n p` □□□□□ `m + p ≤ n + p` □□ `p + n ≤ p + m`□ □□□□□ `+-moroʳ-≤ p m n m≤n` □□□

□□□□□□□□□□□□□

```
+-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)
```

□□ `+-monoˡ-≤ m n p m≤n` □□□ `m + p ≤ n + p`□ □□ `+-monoʳ-≤ n p q p≤q` □□□ `n + p ≤ n + q` □□□□□□□□□□ □□□□ `m + p ≤ n + q` □□□□□□□

□□ `*-mono-≤` □□□

□□□□□□□□□□□

```
-- □□□□□□
```

□□□□

□□□□□□□□□□□□□□□□□□□□

```
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
```

□□□□□□□□□□□□□□□□□□□□0 □□□□□□□□□□□□□ 0□

□□□□□□□□□□□□□□□□□□□□□□**Irreflexive**□□□□ `n < n` □□ □□□ `n` □□
□□□□□□□□□□□□Trichotomy□□□□□□□ `m` □ `n`□`m < n`□`m ≡ n` □□ `m > n` □□□□□□□□□□□□□□□□ `m > n`
□□□□ `n < m` □□□□□□ □□□□□□□□□□□□□□□□□□□

□□ □□□□□□□□□□□ Negation □□□□□□□□

□□□□□□□□□□ `suc m ≤ n` □□□ `m < n`□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `<-trans` □□□□

□□□□□□□□□□□

```
-- □□□□□□□□
```

□□ `trichotomy` □□□□

□□□□□□□□□□□□□□□□□□□□□□□ `m` □ `n` □□□□□□□□□□□

- `m < n`□
- `m ≡ n`□□□
- `m > n`□

□□ `m > n` □ `n < m`□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□
```

□□ `+-mono-<` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□
```

□□ `≤-iff-<` **(□□)**

□□ `suc m ≤ n` □□□ `m < n` □□□□□□□

```
-- □□□□□□□□□□
```

□□ `<-trans-revisited` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□

```
-- □□□□□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□**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)
```

□□□□□□□□ 0□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□ `even` □ `odd` □□□ `where` □□□□□□□□□ □□□□□□□□□□ `ℕ → Set` □□□□□□□□□

□□□□□□□ □□**Overloaded**□□□□□□□□□□□□□ □□□□□□□□ `suc` □□□□□□□□□□

```
suc : ℕ → ℕ

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

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

□□ `zero` □□□□□□□□□□□□□□□Agda □□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□

```
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)
```

□□ □□□□□□□□□□□□□□□

□□□ □□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ 0□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□

□□ `o+o≡e` (□□)

□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□ `Bin-predicates` (□□)

□□□□□□□ Bin □□□□□□□□□□ `Bin` □□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□ 0□□□□11 □□□□□□□□□□

```
⟨⟩ I O I I
⟨⟩ O O I O I I
```

□□□□□□

```
Can : Bin → Set
```

□□□□□□□□□□□□□□□□□□Canonical□□□□□□□□□□□□□ 0□□□□ 11 □□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
One : Bin → Set
```

□□□□□□□□□□□□□□ 1 □□□□□□□□□□□□□□□□□□□□□□□ 1 □□□□□□□□□ □□□□□□□ 0 □□□ 0□□

□□□□□□□□□□□□

```
Can b
------------
```

```
Can (inc b)
```

[illegible]

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

[illegible]

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

[illegible] `One` [illegible] `One b` [illegible] `1` [illegible] `from b` [illegible]

```
-- [illegible]
```

[illegible]

[illegible]

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

[illegible] `≤-total` [illegible] Connectives [illegible] `+-monoʳ-≤` [illegible] `+-monoˡ-≤` [illegible] `+-mono-≤` [illegible] [illegible]

Unicode

[illegible] Unicode [illegible]

```
≤  U+2264  [illegible] (\<=, \le)
≥  U+2265  [illegible] (\>=, \ge)
ˡ  U+02E1  [illegible] L [illegible] (\^l)
ʳ  U+02B3  [illegible] R [illegible] (\^r)
```

`\^l` [illegible] `\^r` [illegible] `l` [illegible] `r` [illegible]

Chapter 4

Equality: □□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□ `A` □□□□ `M` □ `N`□ □□□ `M ≡ N` □□□ `M` □ `N` □□□□□□□□□□□□ □□

□□

□□□□□□□□□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□

□□□

□□□□□□□□□□

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

□□□□□□□□□□□□□□ `A` □□□ `A` □□□ `x` □□□□ `refl` □□□ `x ≡ x` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□ `_≡_` □□□□□□□Argument□□ `x : A` □□□ □□□□□□□Argument□□□ `A → Set` □□□□□□ □□□□□□□□□□□□□□Parameter□□□□□□□ `_≡_` □□□□□□□Argument□ □□□□□□□□□Parameter□□□□□□□□□□□□□□□□Argument□□□□□□□□□□ □□□□□□□□□□□□□

□□□□□□□□□□□□□□

```
infix 4 _≡_
```

□□□ `_≡_` □□□□□□□ 4□□ `_≤_` □□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□ `x ≡ y ≡ z` □□□□□□

□□□□□□□□□□Equivalence Relation□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `refl` □□□□□□□□□□□□□ □□□□□□□□□□□□□□

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

□□□□□□□□□□□□sym □□□□□□ x ≡ y□□□□□□□□□□ refl □□□□□□□ □□□ x □ y □□□□□□□□□□□□□□□□□□□□
x ≡ x □□□□ refl □□□

□□□□□□ sym □□

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

□□□□□□□□□□□□ C-c C-, □Agda □□□□□□

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

□□□□□□□□□□ C-c C-c e□Agda □□ e □□□□□□□□□□□□□□□ □□□□□□□□□□

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

□□□□□□□□□□□□□□□□□ C-c C-, □□□ Agda □□□□□□□□

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

□□□□□□□□□—— Agda □□□ x □ y □□□□□□□□□□ refl □□□□

□□□□□□□□□□□□ C-c C-r □Agda □□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□

□□□□□□□

□□□□□ □□□□Congruence□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□

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

□□□□□□□□□□□□□□

```
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
```

□□

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

□□□□□□□□□□Substitution□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□

□□□ `≡-Reasoning` □□□□□□ Agda □□□□□□□□□□□

```
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
```

□□□□□□□□□□□□□□□□□□□□□□□□□ `module` □□□□□□□□□□□□□□□□□ □□□ `where` □□□
□□□ □□□Open□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□

```
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
  ∎
```

□□□□□□□□□□□□□□□□□□□□

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

□□ `begin` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `_≡⟨_⟩_` □□□ `x` □ `x≡y` □ `y ≡⟨ y≡z ⟩ (z ∎)` □□□□□□□□□□□ `x` □ □□□□□□□□□□□□□□□□□ `x ≡ y` □ `y ≡ z` □□□□□□□ `_≡⟨_⟩_` □□□□□ `trans` □□□□□□□ `x ≡ z` □□□□ `y ≡ z` □□□□□□ `_≡⟨_⟩_` □□□ `y` □ `y≡z` □ `z ∎` □□□□□□□□□□□□□ `y` □□□□□□□□□□□□□□□□ `y ≡ z` □ `z ≡ z` □□□□ □□□ `_≡⟨_⟩_` □□□□□ `trans` □□□□□□□ `y ≡ z` □□□□□□□ `z ≡ z` □□□□□□ `_∎` □□□ `z` □□□□□□ `refl` □□□□□□□□□□□□□□□□□

```
trans x≡y (trans y≡z refl)
```

[illegible] `trans` [illegible] `∎` [illegible] `trans` [illegible] `trans e refl` [illegible] `e` [illegible] `e` [illegible]

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
```

[illegible]

[illegible] [illegible]

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

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

[illegible]

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

[illegible]Postulate[illegible] [illegible]
[illegible]

[illegible]

```
+-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
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□□□□ `_≡⟨⟩_` □□□□□□□ `_≡⟨⟩_` □ `_≡⟨ refl ⟩_` □□□□□

Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□

```
  suc (n + m)
≡⟨⟩
  suc n + m
```

□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
  suc n + m
≡⟨⟩
  suc (n + m)
```

□ Agda □□□□□□Agda □□□□□ `≡⟨⟩` □□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

□□ `≤-Reasoning` **(□□)**

Relations □□□□□□□□□□□□□□□□ `≡-Reasoning` □□□□□□□□□□□□□ □□□□□□ `≤-Reasoning` □□□□□□□□□□□□□□□□□□□□□□□□ `+-monoˡ-≤` □ `+-monoʳ-≤` □ `+-mono-≤` □

```
-- □□□□□□□□□
```

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
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)
```

□□□□□□□□□□□□□□□□□□□□□□□ `even (m + n)` □□□□□□□□□□□□□□□□ `even (n + m)` □□□□□□

Agda □□□□□□□□□□□□□——□□□□□□□□ `rewrite` □□□□□□□□□□□ □□□□□□□□□□□□□ Agda □□□□□□□□□□

```
{-# BUILTIN EQUALITY _≡_ #-}
```

□□□□□□□□□□□□□□□□

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

□□□□ `ev` □□□□□ `even (m + n)` □□□□□□□□□□□□□□ `even (n + m)` □□□□□□□□□□□□□ `rewrite` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□ `even-comm` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□ `C-c C-,` □Agda □□□□

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

□□□□□□□□□

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

□□□□□□□□□□□□□ `C-c C-,` □Agda □□□□□□

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

□□□□□□□□□□□□ `ev` □□□□□□□□□□ `C-c C-a` □□ `ev` □□□□□□□ □□ `C-c C-a` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□

□□

```
+-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
```

□□□□□□□□□□□□□□□□ cong suc (+-comm m n) □□□□□□□□□□□□□ □□□□□□□ +-comm m n □□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□

rewrite □□□□□□ with □□□□□□□□

```
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
```

□□□□□□□□□□ with □□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ m + n □ n + m □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□Dot Pattern□□ .(n + m) □□□□□□□□□□□□□□□□□□□□□□ □□□m + n □ n + m □□□□ +-comm m n □ refl □□ □□□□ Agda □□□□□□□□□——□□□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□

□□□

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

□□□□□□□□ Agda □□

□□□□□Leibniz□□□□

□□□□□□□□□□□□□□□□ Martin-Löf□□ 1975 □□□□□□□□□□□□□□□□□□□□□□□ □ 1686 □□□□□□□□□□□□□□□□□□□□□□□□□□□□Identity of Indiscernibles□□ □□Leibniz’ Law□□ □□□□□□□□□□□“□□□□□□□□□□□□□□□□□”□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□ Martin-Löf □□□□

□□□□□□□□□□□□□□□□□x ≐ y □□□□□ x □□□□□ P □□ y □□□□□□□□ □□□ y □□□□□ P □□ x □□□□

□ x □ y □□□ A □□□□□□□□ x ≐ y □□□□□□□□□□□□ A □□□□□ P □ □□□ P x □□□ P y □

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

□□□□□□□□□□ x ≐ y □□□□□□□□□□ _≐_ {A} x y □□□□□□□ A □□□ A □□□□□□□□□□

□□□□□□□□□□□□Levels□□□□□□□□ Set □□□□ Set □□□□□□□□□□□□□□ □□□□□□□Russell’s Paradox□□□ Girard

□□□□□□□□□□□□□□□□□□□□□□□□□ `Set : Set₁` □ `Set₁ : Set₂` □□□□□□□□□□□□□ `Set` □□□□ `Set₀` □□□□□□□ `_≐_` □□□□□□□□□□□□ `Set` □□□□□□□□□□□□ `Set₁` □□□□□□□□□□□□□□□□□□□□□

□□

```
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)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `P` □ `P x` □□ `P y` □ □□□□□□□□□□□□□□□□

```
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
```

□□ `x ≐ y` □□□□□□ `P` □□□□□□□□□□ `P y` □□ `P x` □□□□ □□□□□□□□□□ `Q` □□□□□□□□□□ `Q z` □ `P z` □□ `P x` □□□□ `Q x` □□□□□□□□□□□□□□□□□□□□□□□□ `x ≐ y` □□□□ `Q y` □□□□ `Q y` □□□□□□□□□□□□□ `P y` □□ `P x` □

□□□□□□□ Martin-Löf □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□ `x ≡ y` □□□□□□□□□□□ `P` □□ `P x` □□□□□□ `P y` □□□□ □□□□□□□□□□□□□□□□□□□□ `x` □ `y` □□□□□□□ `P x` □□□□□ `P y` □□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□ `P` □□□□□□ `P x` □□□□□□ `P y` □□□□ □□□□□□ `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
```

□□□□□□□□□□□□□□□□□□□□□□□□□ `Q` □□□ `Q z` □ `x ≡ z` □□□□□□ □□ `Q x` □□□□□□ Martin Löf □□□□□□□□□□□□ `Q y` □ `x ≐ y` □□□ □ `Q y` □□□□□□□□ `x ≡ y` □□□□

□□□□□□□□□□□□□□□ ≐≃≡: *Leibniz Equality is Isomorphic to Martin-Löf Identity, Parametrically* □□□Andreas Abel□Jesper Cockx□Dominique Devries□Andreas Nuyts □ Philip Wadler□ □□□2017□

□□□□

□□□□□□□□□□□□□□□□□□□□□ `Set` □□□□□□□□□□□□□□□□□□ `Set₀` □ `Set₁` □ `Set₂` □□□□□ `Set` □ `Set₀` □□□□□□ `Set₀ : Set₁` □ `Set₁ : Set₂` □□□□□□ □□□□□□□□□□□ `Set` □□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□ `ℓ` □□□□□ `Set ℓ` □□□□□□□□□□□

□□□□□□□□Universe Polymorphism□□□□□□□□□□□□□□ `ℓ` □□□□ □□□□□□□□□□□□□□□□□□

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

□□□□□□ `zero` □ `suc` □□□□ `lzero` □ `lsuc` □□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□

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

`Set₀` □ `Set₁` □ `Set₂` □□□□□□□□□□□

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

□□□□□□□□□□□□□□□

```
_⊔_ : Level → Level → Level
```

□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□

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

□□□□□□□□□ `Set₁` □□□□□□□□□ `Set` □□□□□□□□□□□□ `Set (lsuc ℓ)` □□□□□□□□□ `Set ℓ` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
_∘_ : ∀ {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)
```

□□□□□□□□□□□□Agda □□□□□□

□□□

□□□□□□□□□□□□□□□□□□□Agda □□□□ `_≡⟨_⟩_` □□□ `step-≡`□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `step-≡` □□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Unicode

□□□□□□□ Unicode□

```
≡  U+2261  □□□ (\==, \equiv)
⟨  U+27E8  □□□□□□ (\<)
⟩  U+27E9  □□□□□□ (\>)
∎  U+220E  □□ (\qed)
≐  U+2250  □□□□□ (\.=)
ℓ  U+2113  □□□□ L (\ell)
⊔  U+2294  □□□□□□□ (\lub)
₀  U+2080  □□ 0 (\_0)
₁  U+2081  □□ 1 (\_1)
₂  U+2082  □□ 2 (\_2)
```

Chapter 5

Isomorphism: □□□□□

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

□□□□□□Isomorphism□□□□Embedding□□ □□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□

```
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 □□□

□□□□□□□□□□□□□□lambda □□□□□□□□□□□□

Lambda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
λ{ P₁ → N₁; ⋯ ; Pₙ → Nₙ }
```

□□□□□□□ `f` □□□□□□

```
f P₁ = N₁
⋯
f Pₙ = Nₙ
```

□□ `Pₙ` □□□□□□□□□□ `Nₙ` □□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□

```
λ x → N
```

□□

```
λ (x : A) → N
```

□□□ `λ{x → N}` □□□□□□□□□□□□

□□□□□ lambda □□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ □Function Composition□

□□□□□□□□□□

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

`g ∘ f` □□□□□□□□ `f` □□□□□ `g` □ □□□□□□□ lambda □□□□□

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

□□□□□Extensionality□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ `cong-app` □□□□□□□□□□□

Agda □□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□ Agda □□□□□□□□□

□□□□□□□□□□□□□□□□□□□□ Naturals □□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□

```
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)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□

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

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

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

□□□Isomorphism□

□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□

```
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 _≃_
```

□□□□□□□□□□□□□□□□□ `A` □ `B` □□□□□□□□□□□□ + □ `A` □ `B` □□□ `to` + □ `B` □□ `A` □□□ `from` + `from` □ `to` □□□□left-inverse□□□□ `from∘to` + `from` □ `to` □□□□right-inverse□□□□ `to∘from`

□□□□□□□□□□□ `from ∘ to` □□□□□□□□□□□□ `to ∘ from` □□□□□□ □□□□□□□□□□□□ `open _≃_` □□ `to` □ `from` □ `from∘to` □ `to∘from` □□□□□□□□□□□□□□□□□□□□□ `_≃_.to` □□□□

□□□□□□□□□□□□□**Record**□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□ Connectives □□□□□□□

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

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

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

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

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

□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□

```
mk-≃′ f g g∘f f∘g
```

□□ `f` □ `g` □ `g∘f` □ `f∘g` □□□□□□□□□

□□□□□□□

□□□ □□ `to` □ `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}
    }
```

□□□ `to` □ `from` □□□□□□□□ `from∘to` □ `to∘from` □□□□□□□□□□ `refl` □□□□□□□□□□□□□ `refl` □□□□□□□□□□□□ `from (to x)` □□□ `x` □□□□□□□□□□

□□□□□□□□□□□□□□ `to` □ `from` □ `from∘to` □ `to∘from` □□□

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

□□□□□□□□□□□□□□ `to` □ `from` □□□□□□□□□□□□□□□□□□□□□□□□□□

```
≃-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
        ∎}
      }
```

□□□□□□□□

□□ □□□□ `_≡⟨⟩_` □□□□□□□□□□□□□□□□□□□□□□□□□

```
module ≃-Reasoning where

  infix  1 ≃-begin_
  infixr 2 _≃⟨_⟩_
  infix  3 _≃-∎

  ≃-begin_ : ∀ {A B : Set}
    → A ≃ B
      -----
    → A ≃ B
  ≃-begin A≃B = A≃B

  _≃⟨_⟩_ : ∀ (A : Set) {B C : Set}
    → A ≃ B
    → B ≃ C
      -----
    → A ≃ C
  A ≃⟨ A≃B ⟩ B≃C = ≃-trans A≃B B≃C

  _≃-∎ : ∀ (A : Set)
      -----
    → A ≃ A
  A ≃-∎ = ≃-refl

open ≃-Reasoning
```

□□□Embedding□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□

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

□□□□□□ `to∘from` □□□□□□□□□□□□□□□□□□□□ `from` □ `to` □□□□□□ `from` □□ `to` □□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
≲-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
        ∎}
     }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□

```
≲-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
        ∎}
    }
```

□□□□□□□□□□□□□□□□□□□□□□□□□ `B ≲ A` □□□□□ □□□□□□ `to` □ `from` □□□□□□□□□□□□□□□

□□□□□□□□

□□□□□□□□□□□□□□□□□□□□

```
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
```

□□ `≃-implies-≲` □□□□

□□□□□□□□□□□□□□

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

```
-- □□□□□□□□□
```

□□ `_⇔_` □□□□

□□□□□□□□□□□□□□□□“□□□□“□□

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

□□□□□□□□□□□□□□□□

```
-- □□□□□□□□
```

□□ `Bin-embedding` □□□□

□□□□ Bin □ Bin-laws□ □□□□□□□□□□□□ `Bin` □□□□□□□□□□□□□□□□□□□

```
to : ℕ → Bin
from : Bin → ℕ
```

□□□□□□□□□

```
from (to n) ≡ n
```

□□□□□□□□□□□□□□ `ℕ` □ `Bin` □□□□

```
-- □□□□□□□□
```

□□□ `to` □ `from` □□□□□□□□□

□□□

□□□□□□□□□□□□□□□□□□□

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

□□□□□ `_↔_` □ `_↞_` □□□□□□□□□□ `_≃_` □ `_≲_`□ □□□ □□□□□□□□□□□□□□□□□

Unicode

□□□□□□□□□□ Unicode□

```
∘  U+2218  □□□□□ (\o, \circ, \comp)
λ  U+03BB  □□□□□□□ LAMBDA (\lambda, \Gl)
≃  U+2243  □□□□□ (\~-)
```

```
≲  U+2272  □□□□□□ (\<~)
⇔  U+21D4  □□□□□ (\<=>)
```

Chapter 6

Connectives: □□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□Propositions as Types□□

- □□□Conjunction□□□□□Product□
- □□□Disjunction□□□□□Sum□
- □□True□□□□□□□□□Unit Type□
- □□False□□□□□□□□Empty Type□
- □□□Implication□□□□□□□□□Function Space□

□□

```
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
```

□□□□□

□□□□□□ `A` □ `B`□□□□ `A × B` □□□ `A` □□□ `B` □□□ □□□□□□□□□□□□□□□□□□□□□□

```
data _×_ (A B : Set) : Set where

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

`A × B` □□□□□□ `⟨ M , N ⟩` □□□□□□□□ `M` □ `A` □□□□□□ `N` □ `B` □□□□□□

□□ `A × B` □□□□□□□□□□□□ `A` □□□ `B` □□□

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

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

□□ `L` □ `A × B` □□□□□, □□ `proj₁ L` □ `A` □□□□□□ `proj₂ L` □ `B` □□□□□□

□ `⟨_,_⟩` □□□□□□□□□□□□□□□□□□□□□□□□□Constructor□□ □□□□□□□□□□□□□□□□□□□□□Destructor□□□□□□□ `proj₁` □ `proj₂` □□□□□□□□□□□□□□□□□□

□□□□□□ `⟨_,_⟩` □□□□□Introduce□□□□□ `proj₁` □ `proj₂` □□□□□Eliminate□□□□ □□□□□ `×-I` □□□ `×-E₁` □ `×-E₂` □□□□□□□□□□□□□□□□□□□□□□ □□□ □□ □□□□□□□□□□□——□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□——□□□□□□□□□□□□[1]

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□ `⟨ x , y ⟩` □□□ `w` □□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
infixr 2 _×_
```

□□□ `m ≤ n × n ≤ p` □□□ `(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 `_x_` and the record type `_×′_` behave similarly. One difference is that for data types we have to prove η-equality, but for record types, η-equality holds *by definition*. While proving `η-×′`, we do not have to pattern match on `w` to know that η-equality holds:

[1] □□□□□ Propositions as Types□□□□□□□□□□□□ □□□Philip Wadler□□□□ □ACM □□□2015 □ 9 □

```
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.

□□□□□□ `A` □ `B` □□□□ `A × B` □□ `A` □ `B` □□□ □□□□□□□□□□□□□□□□□Cartesian Product□□□□□□□□□□□□□□□□□□□□□□□ `A` □ `m` □□□□□□□□□□ `B` □ `n` □□□□□□□□ □□□□ `A × B` □ `m * n` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `Bool` □□□□□□□□□□□ `Tri` □□□

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

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

□□□ `Bool × Tri` □□□□□□□□□□□□

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

□□□□□□□□□□□□□□ `Bool × Tri` □□□□

```
×-count : Bool × Tri → ℕ
×-count ⟨ true , aa ⟩  = 1
×-count ⟨ true , bb ⟩  = 2
×-count ⟨ true , cc ⟩  = 3
×-count ⟨ false , aa ⟩ = 4
×-count ⟨ false , bb ⟩ = 5
×-count ⟨ false , cc ⟩ = 6
```

□□□□□□□□□□□□□□□□——□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□ `to` □□□□□□□□□□□ `⟨ x , y ⟩` □□ `⟨ y , x ⟩`□ `from` □□□□□□□□□□□□□□ □ `from∘to` □ `to∘from` □□□□□□□□□□□□□□□□□□□□ □□ `λ w → refl` □□ `from∘to` □□□□□□□□□ `to∘from` □□□

```
×-comm : ∀ {A B : Set} → A × B ≃ B × A
×-comm =
  record
    { to      = λ{ ⟨ x , y ⟩ → ⟨ y , x ⟩ }
    ; from    = λ{ ⟨ y , x ⟩ → ⟨ x , y ⟩ }
    ; from∘to = λ{ ⟨ x , y ⟩ → refl }
    ; to∘from = λ{ ⟨ y , x ⟩ → refl }
    }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
m * n ≡ n * m
A × B ≃ B × A
```

□□□□□□□□□□□□□ `m` □ `2`□`n` □ `3`□□□ `m * n` □ `n * m` □□ `6`□ □□□□□□□□□□□□□ `A` □ `Bool` □ `B` □ `Tri`□□□ `Bool × Tri` □ `Tri × Bool` □□□□□□□□□□□□□□□□□□□□□□□□`⟨ true , aa ⟩` □□□□□□□□□□□□□□□ `⟨ aa , true ⟩`□

□□□□□□□□ `to` □□□□□□□□□□□□□□ `⟨ ⟨ x , y ⟩ , z ⟩` □□□ `⟨ x , ⟨ y , z ⟩ ⟩`□ `from` □□

```
×-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 }
    }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□ `(ℕ × Bool) × Tri` □ `ℕ × (Bool × Tri)` □□□□□□□□□□□ □□□□□□□ `⟨ ⟨ 1 , true ⟩ , aa ⟩` □□□□□□□□□□ `⟨ 1 , ⟨ true , aa ⟩ ⟩`□ □□□□□□□□□□□□

□□ `⇔≃×` □□□□

□□□□□□□ `A ⇔ B` □ `(A → B) × (B → A)` □□□

```
-- □□□□□□□□
```

□□□□□□□□

□□ `⊤` □□□□□□□□□□□□□□□□□□□□□□□□

```
data ⊤ : Set where

  tt :
    --
    ⊤
```

`⊤` □□□□□□ `tt` □□□□□□

□□□□□□□□□□□□□□□□□□□□ `⊤` □□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□

`η-×` □ □□□□□ `η-⊤`□□□□□□□ `⊤` □□□□□□□□ `tt`□

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

□□□□□□□□□□□□□□ `w` □□□ `tt` □□□□□□□□□□□□□□□

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′ = _
```

□□□ `⊤` □□□□□□□Unit Type□□□□□□ `⊤` □□□□□□□□ `tt` □ □□□□□□□□□□□□□□ `⊤` □□□□□□□

->

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

□□□□□□1 □□□ `to` □□□ `⟨ tt , x ⟩` □□□ `x` □ `from` □□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□

```
⊤-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 }
    }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
1 * m ≡ m
⊤ × A ≃ A
```

□□□□□□□□□□□□□□□ `m` □ `2` □□□ `1 * m` □ `m` □□ `2` □ □□□□□□□□□□□□□□□□ `A` □ `Bool` □□□ `⊤ × Bool` □ `Bool` □□□□□ □□□ `⟨ tt , true ⟩` □□□□□□□□□□□□□□□□□ `true` □

□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□

□□□

□□□□ `A` □ `B` □□ `A ⊎ B` □ `A` □□□ `B` □□□□ □□□□□□□□□□□□□□

```
data _⊎_ (A B : Set) : Set where

  inj₁ :
    A
    -----
    → A ⊎ B

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

`A ⊎ B` □□□□□□□ `inj₁ M` □□ `M` □ `A` □□□□□ `inj₂ N` □□ `N` □ `B` □□□□

□□ `A → C` □ `B → C` □□□□□□□□ `A ⊎ B` □□□□□□□ `C` □□

```
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
```

□ `inj₁` □ `inj₂` □□□□□□□□□□□□□□□□□

□ `inj₁` □ `inj₂` □□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□ `case-⊎` □□□□□□□□□□□□□□□□ `inj₁` □ `inj₂` □□□□□ □ `case-⊎` □□□□□□□□□ `⊎-I₁` □ `⊎-I₂` □□ `⊎-E` □

□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□

```
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
```

□□□□□□□□□ `inj₁ x` □□ `w` □□□□□□□□□□□ `inj₂ y` □□

□□□□□□□□□□□□□□□□□□□□□□

```
infixr 1 _⊎_
```

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

□□□□□□□ `A` □ `B` □□□□ `A ⊎ B` □□ `A` □ `B` □□□ □□□□□□□□□□□□□□□□Disjoint Union□□□□□□□□□□□□□□□□□□□□□□□□□□ `A` □ `m` □□□□□□□□□ `B` □ `n` □□□□□□□□ □□□□ `A ⊎ B` □ `m + n` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `Bool` □□□□□□□□□□ `Tri` □□□□□□□□□□□ □□□ `Bool ⊎ Tri` □□□□□□□□□□□

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

□□□□□□□□□□□□□ `Bool ⊎ Tri` □□□□

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

□□□□□□□□□□□□□□□□——□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `⊎-comm` □□□□

□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□□
```

□□ `⊎-assoc` □□□□

□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□□
```

□□□□□□

□□ `⊥` □□□□□□□□□□□□□□□□□□□□□□□□□□□

```
data ⊥ : Set where
  -- □□□□□
```

□□ `⊥` □□□□□□

□ `⊤` □□□□ `⊥` □□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□ `⊥` □□□ *ex falso* □□□□□□□□□□□ “□□□□□□□□□□□□□□□□□”□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□Absurd Pattern□ `()` □□□□□□□ `⊥` □□□□□□□□□□□□□□□ `()` □□□□□□□□□□□□□□□□□□□□□□□

`case-⊎` □□□□□□ `⊥-elim` □□□□□□□□□□□□□□ `case-⊥` □ □□□□□□□□□□□□□□□□□□□□□

`uniq-⊎` □□□□□□ `uniq-⊥` □□□□□□ `⊥-elim` □□□□ `⊥` □□□□□□□□□

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

□□□□□□□□□□ `w` □□□□□□□□□□□□□□□□□□□□□□

```
⊥-count : ⊥ → ℕ
⊥-count ()
```

□□□□□□□□□□□□□□□□□□□□□□□□□ `⊥` □

□□□□□□□0 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `⊥-identityˡ` □□□□

□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□□
```

Exercise `⊥-identityʳ` (practice)

□□ `⊥-identityʳ` □□□□

□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□□
```

□□□□□□

□□□□□□□ `A` □ `B` □□□□ `A → B` □□□ `A` □□□□□□`B` □□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

`A → B` □□□□□□□□□□□□□□□□□

```
λ (x : A) → N
```

□□ `N` □□□□□□ `B` □□□□□□□□□□□□□ `A` □□□□□ `x` □ □□□□ `A → B` □□□□□ `L` □□□□ `A` □□□□□ `M` □□□ `L M` □ `B` □□□□□□ □□□□□ `A → B` □□□□□□□□□□□□ `A` □□□□□□□□ `B` □□□□□□

□□□□□□□□□□ `A → B` □ `A` □□□□□□□□□□□□□□ `B` □□□

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

```
→-elim L M = L M
```

□□□□□□□□□□□□□ *modus ponens*□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□ Lambda □□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□ `A ⊎ B → B ⊎ A` □□□□□ `(A ⊎ B) → (B ⊎ A)`□

□□□□□□□ `A` □ `B` □□□□ `A → B` □□□ `A` □ `B` □□□□□□□ □□□□□□□□□ `B` □□□ `A` □□□□□□□□□□□ `A` □ `m` □□□□□□□ □□ `B` □ `n` □□□□□□□□□□□□□ `A → B` □ n^m □□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `Bool` □□□□□□□□□□□ `Tri` □□□ □□□□□□□□□□□□□ `Bool → Tri` □□□□□□□□□□□□□□□□□□□□

```
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}
```

□□□□□□□□□□□□□□□ `Bool → Tri` □□□□

```
→-count : (Bool → Tri) → ℕ
→-count f with f true | f false
... | aa | aa = 1
... | aa | bb = 2
... | aa | cc = 3
... | bb | aa = 4
... | bb | bb = 5
... | bb | cc = 6
... | cc | aa = 7
... | cc | bb = 8
... | cc | cc = 9
```

□□

□□□□□□□□□

```
(p ^ n) ^ m  ≡  p ^ (n * m)
```

□□□□□□□□□□

```
A → (B → C)  ≃  (A × B) → C
```

□□□□□□□□□□□□□ `A` □□□□□□□ `B` □□□□□□□□□ `C` □□□□□□□ □□□□□□□□□□□□□□□□Currying□□□□□□□□□□□□□□□

```
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 }}
    }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□ □□□□□□□□□□□□□□□□□□□

```
_+_ : ℕ → ℕ → ℕ
```

□□□□□□□□□□□□□□□□□□□□□□□□□□

```
_+′_ : (ℕ × ℕ) → ℕ
```

Agda □□□□□□□□□□□□ `2 + 3` □ `_+_ 2 3` □□□□□□□□□□□□□□□□□□□□ `2 +′ 3` □□□ `_+′_ ⟨ 2 , 3 ⟩` □□□□

□□□□□□□□

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

□□□□□□□□□

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

□□□□ `A` □□□ `B` □□□□ `C` □□□□□□□□ `A` □□□□ `C` □□□ □□ `B` □□□□ `C` □□□□□□□□□□□□□□□□□□□□

```
→-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 }
    }
```

□□□□□□□□

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

□□□□□□□□□

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

□□□□ `A` □□□□ `B` □□ `C` □□□□□□□□ `A` □□□□ `B` □□□ □□ `A` □□□□ `C` □□□□□□□□□□□□□□□□□□□□□ `η-×` □□□

```
→-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 }
    }
```

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
×-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
               }
    }
```

□□□□□□□□□□□□□□□□□

```
⊎-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
               }
    }
```

□□□□□ `from` □□□□□□□□□□□□□□□□□□□□ `⟨ inj₂ z , inj₂ z′ ⟩` □□□ `inj₂ z` □□□□□□□□□ `inj₂ z′` □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□ `from` □□□□□□ `z` □□ `z′` □□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□”□□“□

□□ `⊎-weak-×` □□□□

□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□
```

□□ `⊎×-implies-×⊎` □□□□

□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□□□

```
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 (_⇔_)
```

□□□□□□ `_,_` □□□□□□□□□□□ `⟨_,_⟩`□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□ `a , b , c` □□ `(a, (b , c))` □□□□□□□□□□□□□□□□□□ □□□ Lists □□ `[_,_]` □□□□□□□□□□□□ □□ DeBruijn □□□□ `Γ , A` □□□□□□□□□ □□□□□ `_⇔_` □□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□

Unicode

□□□□□□□ Unicode□

```
×  U+00D7  □□□□ (\x)
⊎  U+228E  □□□□□ (\u+)
⊤  U+22A4  □□□□ (\top)
⊥  U+22A5  □□□□ (\bot)
η  U+03B7  □□□□□□ ETA (\eta)
₁  U+2081  □□ 1 (\_1)
₂  U+2082  □□ 2 (\_2)
⇔  U+21D4  □□□□□ (\<=>)
```

Chapter 7

Negation: [illegible]

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

[illegible]

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)
```

[illegible]

[illegible] `A` [illegible] `A` [illegible] `¬ A` [illegible]

```
¬_ : Set → Set
¬ A = A → ⊥
```

[illegible]**Reductio ad Absurdum**[illegible] `A` [illegible] `⊥` [illegible] `¬ A` [illegible]

`¬ A` [illegible]

```
λ{ x → N }
```

[illegible] `N` [illegible] `⊥` [illegible] `A` [illegible] `x` [illegible] `¬ A` [illegible] `A` [illegible] `⊥` [illegible]

[illegible] `¬ A` [illegible] `A` [illegible] `⊥` [illegible] `¬ A` [illegible] `A` [illegible]

```
¬-elim : ∀ {A : Set}
  → ¬ A
  → A
```

```
    ---
  → ⊥
¬-elim ¬x x = ¬x x
```

□□□□□□□□ `¬ A` □□□□□ `¬x` □□ `A` □□□□□ `x` □□□□ `¬x` □□□□□□ `A → ⊥` □□□□□□□□ `¬x x` □□□□□□□ `⊥` □□□□□□□□ `→-elim` □□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
infix 3 ¬_
```

□□□ `¬ A × ¬ B` □□□□ `(¬ A) × (¬ B)` □□ `¬ m ≡ n` □□□□ `¬ (m ≡ n)` □

□□□□□□□□ `A` □□□ `¬ ¬ A` □□□□□□□Agda □□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□ `A` □□ `¬ ¬ A` □

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

□ `x` □ `A` □□□□□□□□□□□□□□ `¬ A` □□□□□□□□□□□□□□ `¬ ¬ A` □□□□□□□ `¬x` □ `¬ A` □□□□□□□□ `¬x x` □□□□□□ `A` □ `¬ A` □□□□□□□□ □□□□□□□□□□ `¬ ¬ A` □

□□□□□□□□□□□□□□□

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

□□□□□□□□□□ λ-□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□ `¬ ¬ A` □□ `A` □□□□□□□ `¬ ¬ ¬ A` □□ `¬ A` □

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

□ `¬¬¬x` □ `¬ ¬ ¬ A` □□□□□□□□□□□□□□ `A` □□□□□□□□□□ □□ `¬ A` □□□□□□□ `x` □ `A` □□□□□□□□□□□□□□□□ `¬¬-intro x` □□□□□□□□□ `¬ ¬ A` □□□ `¬¬¬x (¬¬-intro x)` □□□□□ `¬ ¬ ¬ A` □ `¬ ¬ A` □□□□□□□□□□□□□□□□ `¬ A` □

□□□□□□□□□□□□□□□□□**contraposition**□□□□□□□□□ `A` □□ `B` □ □ `¬ B` □□ `¬ A` □

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

□ `f` □ `A → B` □□□□ `¬y` □ `¬ B` □□□□□□□□□□□□□□□□ `A` □□□□□□□□□□□□□□□ `¬ A` □□□□□□□ `x` □ `A` □□□□□□□ `f x` □

□□□ A → B □ A □□□□□□ B □□□ ¬y (f x) □□□ B □ ¬ B □□□ ⊥ □□□□□□□□ ¬ A □

□□□□□□□□□□□

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

□□□□□□□□□□□□

```
_ : 1 ≢ 2
_ = λ()
```

□□□□□□□ λ-□□□□□□□□Absurd Pattern□□□ M ≡ N □□□ M □ N □□□□□□□□□□□□□□ 1 □ 2 □□□□□□□□□□□□□□□□ Agda □□□□□□□□ 1 ≡ 2 □ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□ zero ≡ suc m □□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□ ⊥ → ⊥ □□□□□□□□□□□□□□□□

```
id : ⊥ → ⊥
id x = x

id′ : ⊥ → ⊥
id′ ()
```

□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□ x □□□ id x ≡ id′ x □ □ id ≡ id′ □□□□□□ x □□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□

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

¬ A □□□□□□ A □□□□□□□□□□□□□□□□□□□□ A □□□ x □□□□□□□ x □□□□□□□□□□□□□□□□□

□□ <-irreflexive □□□

□□□□□□□□□□□□□□□ □ n < n □□□ n □□□□

```
-- □□□□□□□
```

[illegible] `trichotomy` [illegible]

[illegible][illegible] [illegible] `m` [illegible] `n` [illegible]

- `m < n`
- `m ≡ n`
- `m > n`

[illegible] [illegible]

```
-- [illegible]
```

[illegible] `⊎-dual-×` [illegible]

[illegible]De Morgan's Law[illegible]

```
¬ (A ⊎ B) ≃ (¬ A) × (¬ B)
```

[illegible]

```
-- [illegible]
```

[illegible]

```
¬ (A × B) ≃ (¬ A) ⊎ (¬ B)
```

[illegible]

[illegible]

[illegible] Gilbert [illegible] Sullivan [illegible]*The Gondoliers*[illegible] Casilda [illegible] Marco [illegible] Giuseppe [illegible]

[illegible] [illegible] [illegible] `A ⊎ B` [illegible] `A` [illegible] `B` [illegible] [illegible] Casilda [illegible] Marco [illegible] Giuseppe[illegible] Gilbert [illegible] Sullivan [illegible] [illegible] Luiz[illegible] Casilda [illegible]

[illegible]Law of the Excluded Middle[illegible]————[illegible] `A`[illegible]`A ⊎ ¬ A` [illegible]———— [illegible] `A` [illegible] `¬ A` [illegible] [illegible]Heyting[illegible]Hilbert[illegible] [illegible]Kolmogorov[illegible] [illegible]

[illegible] [illegible] `A ⊎ B` [illegible] `A` [illegible] `B` [illegible]Disjoint Sum[illegible]

[illegible] "Propositions as Types", Philip Wadler, *Communications of the ACM*[illegible]2015 [illegible] 12 [illegible]

□□□□□□□

□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□**Irrefutable**□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□

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

□□ `¬ (A ⊎ ¬ A)` □□□ `k` □□□□□□□□□□□□□□□□□ `A ⊎ ¬ A` □□□ □□□□□□□□□□□□□□□□□ `?` □□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□ `k` □□□□□□□□□□□□□□□□

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

□□□□□□□□□ `A ⊎ ¬ A` □□□□□□□□□□□□□□□□□□□□□□□□ `A` □□□ □□□□□□□□□□□□

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

□□□□□□□□ `¬ A` □□□□□□□□□□□□□□□□□□ `A` □□□□□□□□□□□□ □□□ `x` □□□□□□□ `A` □□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□ `k` □□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□ `A` □□□□□□□□ `x`□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□ □□ Peter Selinger □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□ (a) □□□ 10 □□□□□□□ (b) □□□□□□□ 10 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ (a) □□ (b)□□

□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□ (b)□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□ (a) □□ (b)□□

□□□□□□□□□□□ (b)□□

□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□ 10 □□□□□□□□□□□□□

□□□ 10 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□ (b) □□□□□□□□□□ (a)□ □□□□□□□ 10 □□□□□

□□□□□□□□□□□□□□□□□□

□□□□□□□□□ “Call-by-Value is Dual to Call-by-Name”, Philip Wadler, *International Conference on Functional Programming*, 2003 □□□

□□ `Classical` □□□□

□□□□□□□

- □□□□□□□□ `A`□`A ⊎ ¬ A`□
- □□□□□□□□□□□□ `A`□`¬ ¬ A → A`□
- □□□□□□□□□□□ `A` □ `B`□`((A → B) → A) → A`□
- □□□□□□□□□□□□□ `A` □ `B`□`(A → B) → ¬ A ⊎ B`□
- □□□□□□□□□□□ `A` □ `B`□`¬ (¬ A × ¬ B) → A ⊎ B`□

□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□
```

□□ `Stable` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□**Stable**□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□

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

Unicode

□□□□□□□ Unicode□

```
¬  U+00AC  □□□□ (\neg)
≢  U+2262  □□□□ (\==n)
```

Chapter 8

Quantifiers: □□□□□□□□□□

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

□□□□□□□□□□Universal Quantification□□□□□□□Existential Quantification□□

□□

```
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)
```

□□□□

□□□□□□□□□□Dependent Function Type□□□□□□□□□□ □□

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

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

□□□□□□ `A` □□□□□ `x` □□□□□ `x` □□□□□□□ `B x`□□□□□ □□□ `∀ (x : A) → B x` □□□□□□□□ `A` □□ `M`□□□ `B M` □□□□□□ □□□□`B M` □□□□ `B x` □□□□□□□□ `x` □□□ `M` □□□□□ □□ `x` □ `B x` □□□□□□□□□□□□□□□ `∀ (x : A) → B x` □□□□□□

`∀ (x : A) → B x` □□□□□□□□□□□□□

```
λ (x : A) → N x
```

□□ `N x` □□□ `B x` □□□□□`N x` □ `B x` □□□□□□ `A` □□□□□□□ `x`□ □□□□□ `L`□ □□□ `∀ (x : A) → B x` □□□□□□□□□□□□ `A` □□ `M`□ `L M` □□□□□ `B M` □□□□□□□□□□□`∀ (x : A) → B x` □□□□□□□□□□□ □□□□ `A`

□□ `M` □□□ `B M` □□□□□□

□□□□□□□□□□□□ `∀ (x : A) → B x` □□□□□□ `M` □□□□□□ `A` □□□ □□□□□□□□□ `B M` □□□

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

□ `→-elim` □□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□

□□□□□□□□□□□□□□Dependent Product□□□□□□ `A` □□□□□□□□□□□ □□ `x₁ , ⋯ , xₙ` □□□□□□□ `B x₁ , ⋯ , B xₙ` □ `m₁ , ⋯ , mₙ` □□□□□□□ □□ `∀ (x : A) → B x` □ `m₁ * ⋯ * mₙ` □□□□□□□ `∀ (x : A) → B x` □□□□□ □□ `Π[ x ∈ A ] (B x)` □□□□□ `Π` □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `∀-distrib-×` □□□□

□□□□□□□□□□□□□□□□

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

□□□□□□ Connectives □□□□ (`→-distrib-×`) □□□□□

□□ `⊎∀-implies-∀⊎` □□□□

□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `∀-×` □□□□

□□□□□□□□

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

□ B □□□ Tri □□□□□□□□□□□□□□ B : Tri → Set □ □□ ∀ (x : Tri) → B x □ B aa × B bb × B cc □□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□

□□□□ A □□□□□ x □□□□□ x □□□□□□□□ B x □□□□□ □□□ Σ[x ∈ A] B x □□□□□□□□□ A □□ M □□□ B M □□□□□□ □□□□ B M □□□□ B x □□□□□□□□□ x □□□ M □□□□□□ □□ x □ B x □□□□□□□□□□□□□□□□□ Σ[x ∈ A] B x □□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□

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

□□□ □□□□□□□□□□□□ Σ-syntax □□□□□□□

Σ[x ∈ A] B x □□□□□□ ⟨ M , N ⟩ □□□□□ M □□□□ A □□□ N □ B M □□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□

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

□□□□

```
⟨ M , N ⟩
```

□□ M □□□□ A □□□N □□□□ B M □□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□ Agda □□□

□□□□□□□□□□□□Dependent Sum□□□□□□ A □□□□□□□□□□□ □□ x₁ , ⋯ , xₙ □□□□□□□ B x₁ , ⋯ , B xₙ □ m₁ , ⋯ , mₙ □□□□□□□ □□ Σ[x ∈ A] B x □ m₁ + ⋯ + mₙ □□□□□□□□□□□□□□□□□□□□□□□□□—— Σ □□□□

□□□□□□□□□□□□□□Dependent Product□□□□□□□□□□□□□□□□□□□□ □□

□□□□□□□□□□ ∃ □□□□□□□ ∀ □□□□□□□□□□□ Agda □□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□

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

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

□□□□□□□□□□ `∃-syntax` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□

□□ `∀ x → B x → C` □□□□□□□□ `C` □□□□□□□ `x` □□□ `∃[ x ] B x` □□□ □□□□□□□□□□ `C` □□□

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

□□□□□□□□□□□□□□□ `A` □□□ `x` □ `B x` □□□ `C` □□□□□□□□□□ `A` □ `x` □ `B x` □□□□□□□□□□□□ `C` □□□□□□□□□□□□□ `∀ x → B x → C` □□□□□ `A` □□□ `x` □ `B x` □□□ `y` □□□□□□□□□□□□□□ `∃[ x ] B x` □□□□□□□

□□□□□□□□□□□□□□□□□□□□□□

```
∀∃-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 }}
    }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□

□□ `∃-distrib-⊎` □□□□

□□□□□□□□□□□□□□□□

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

□□ `∃×-implies-×∃` □□□□

□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `∃-⊎` □□□□

□□□□ `∀-×` □□ `Tri` □ `B` □ □□ `∃[ x ] B x` □ `B aa ⊎ B bb ⊎ B cc` □□□□□

□□□□□□□

□□□□□ Relations □□□□□□ `even` □ `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)
```

□□□□□□ 0 □□

□□□ □□□□□□□□□□□□□□□□□□□□□□□

`even n` □□□□ `∃[ m ] (    m * 2 ≡ n)`

`odd  n` □□□□ `∃[ m ] (1 + m * 2 ≡ n)`

□□ □□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□

```
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 ⟩
```

□□□□□□□□□□□□□□□□ `n` □□□□□□□□□□□□□□□□□□□ `m` □□□ `m * 2 ≡ n` □□ `1 + m * 2 ≡ n` □□□□□□□□ `n` □□□ □□□□□□□□□□□□□

- □□□□□□□□□□□□□ 0□□□□□□□□□□□ 0 □□□ 0 □□□□ 0 □□□□
- □□□□□□□□□□□□□□□□□□□ 1□□□□□□□□□□□□□□□□□□□□□ `m` □ `1 + m * 2 ≡ n` □□□□□□□□□□□ `suc m` □□ `suc m * 2 ≡ suc n` □□□—— □□□□□□□□□□ `n` □□□□□□

- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `m` □ `m * 2 ≡ n` □□□□□□□□□ `suc m` □□ `1 + m * 2 ≡ suc n` □□□—— □□□□□□□□□ `n` □□□□□

□□□□□□□□□□□□

□□□□□□□□□□

```
∃-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 ⟩)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□

- □□□□ `zero` □□□□□□□□ `zero * 2` □□□□ `even-zero` □□□
- □□□□ `suc n` □□□□□□□□ `suc m * 2` □□□□□□□□□□ `1 + m * 2` □□□□□□□□□□ `even-suc` □□□
- □□□□□□□□□□□□ `1 + m * 2` □□□□□□□□□□ `m * 2` □□□□ □□□□□□□ `odd-suc` □□□

□□□□□□□□□□□□□

□□ `∃-even-odd` □□□□

□□□□□ `2 * m` □□ `m * 2` □ `2 * m + 1` □□ `1 + m * 2` □□□□□□□□□□□□ □□□□□□□□ `∃-even` □ `∃-odd` □

```
-- □□□□□□□□□
```

□□ `∃-|-≤` □□□□

□□□□□□□□ `x` □□ `x + y ≡ z` □□□ `y ≤ z` □□□

```
-- □□□□□□□□□
```

□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
¬∃≃∀¬ : ∀ {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 }
    }
```

□ `to` □□□□□□□□□ `¬ ∃[ x ] B x` □□□□ `¬∃xy` □□□□□□□□□ `x` □□□ □□□□□ `¬ B x` □□□□□□□□□□ `B x` □□□□ `y` □□□□□□□□□□□ `x` □ `y` □□□□□□□□□□ `∃[ x ] B x` □□□□ `⟨ x , y ⟩` □□□□□ `¬∃xy` □□□□□□□

□ `from` □□□□□□□□□ `∀ x → ¬ B x` □□□□ `∀¬xy` □□□□□□□□□□□ `∃[ x ] B x` □□□□ `⟨ x , y ⟩` □□□□□□□□□□□ `∀¬xy` □□□ `x` □□□ □□□□□□□ `¬ B x` □□□□□□□ `y` □□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□

□□ `∃¬-implies-¬∀` □□□□

□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ `Bin-isomorphism` □□□□

□□□□□ Bin□ Bin-laws □ Bin-predicates □□ □□□□□□□□□□□□ `Bin` □□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□

```
from (to n) ≡ n

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

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

□□□□□□□ `ℕ` □ `∃[ b ](Can b)` □□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□ `b` □ `One b` □□□□□□□ `Can b` □□□□□□

```
≡One : ∀ {b : Bin} (o o′ : One b) → o ≡ o′

≡Can : ∀ {b : Bin} (cb cb′ : Can b) → cb ≡ cb′
```

Many of the alternatives for proving `to∘from` turn out to be tricky. However, the proof can be straightforward if you use the following lemma, which is a corollary of `≡Can`.

```
proj₁≡→Can≡ : {cb cb′ : ∃[ b ] Can b} → proj₁ cb ≡ proj₁ cb′ → cb ≡ cb′
```

```
-- □□□□□□□□□
```

□□□

□□□□□□□□□□□□□□□

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

Unicode

□□□□□□□ Unicode□

```
Π  U+03A0  □□□□□□ PI (\Pi)
Σ  U+03A3  □□□□□□ SIGMA (\Sigma)
∃  U+2203  □□ (\ex, \exists)
```

Chapter 9

Decidable: □□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□Evidence□□□□□□□□□□ □□□□□□□□□Compute□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□Boolean□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□Decidable□□□□□□□□□□□

□□

```
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 (_⇔_)
```

□□ vs □□

□□□□□ Relations □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
infix 4 _≤_

data _≤_ : ℕ → ℕ → Set where

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

  s≤s : ∀ {m n : ℕ}
    → m ≤ n
```

```
      -------------
    → suc m ≤ suc n
```

□□□□□□□□□□ `2 ≤ 4` □□□□□□□□□□□□□□□□ `4 ≤ 2` □□□□□□□

```
2≤4 : 2 ≤ 4
2≤4 = s≤s (s≤s z≤n)

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

`()` □□□□□□□□□ `2 ≤ 0` □□□□□□□ `z≤n` □□□□□□□□ `2` □□ `zero` □□ `s≤s` □□□□□□□□□ `0` □□□□□ `suc n` □□

□□

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

□□□ `true` □ □□□□□□ `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
```

□□ `m` □□□□□□□ `suc m ≤ zero` □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□ `2 ≤ᵇ 4` □□□□□□□□□ `4 ≤ᵇ 2` □□□□□

```
_ : (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
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ 0□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□ `s≤s` □□□ `z≤n` □□□ `2 ≤ 4`□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ 0□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□ `s≤s` □□□ `()` □□□□□□ `4 ≤ 2` □□□□□

□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□

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

□□□ `⊤` □□□□□□□ `tt` □□□□□□⊥ □□□□□□□□□□□□□ `T` □□□□□ `t`□ □ `⊤` □□□□□□ `b` □ `Bool` □□□□□□□□ `b` □□□ `tt` □□□□ `T b` □□□□□□ □□ `b` □□□□□□□ `T b` □□□□□□

□□□□□ `T b` □□□□ `b ≡ true` □□□□□□□□□□□□□□□□□□□□ `b` □□□□□□□

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

□□ `b` □□□□□ `T b` □ `tt` □□□ `b ≡ true` □ `refl` □□□ □ `b` □□□□□ `T b` □□□□□

□□□□□□□□□□□□□□□ `b` □□□□□□

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

□□ `b ≡ true` □ `refl` □□□□□□□ `b` □ `true` □□□ `T b` □ `tt` □□□

□□□□□□□□ `T (m ≤ᵇ n)` □□□□ `m ≤ n` □□□□□□

□□□□□□□□□□□ `_≤ᵇ_` □□□□□□□□□

```
≤ᵇ→≤ : ∀ (m n : ℕ) → T (m ≤ᵇ n) → m ≤ n
≤ᵇ→≤ zero n       tt   = z≤n
≤ᵇ→≤ (suc m) zero ()
≤ᵇ→≤ (suc m) (suc n) t = s≤s (≤ᵇ→≤ m n t)
```

□□□□□□□□□□□□□□□ `zero ≤ᵇ n` □□□□□ `T (m ≤ᵇ n)` □ `tt` □□□ □□□□ `m ≤ n` □ `z≤n` □□□□□□□□□□□□□□□□□□ `suc m ≤ᵇ zero` □□□□ `T (m ≤ᵇ n)` □□□□□□□□□□□ `m ≤ n` □□□□□□□□□□□□□□□□□□□□□□□□□ `suc m ≤ᵇ suc n` □□□ `m ≤ᵇ n` □□ `t` □ `T (suc m ≤ᵇ suc n)` □□□□□□□□□□□ □□ `_≤ᵇ_` □□□□□□□ `T (m ≤ᵇ n)` □□□□□□□□□□□□□□□□ `m ≤ n` □□□□□□□ `s≤s` □□□□□□ `suc m ≤ suc n` □□□□

□□□□□□□□□□□ `m ≤ n` □□□□□□□□□□

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

□□□□□ `z≤n` □□□□□□□□□ `zero ≤ᵇ n` □□□□□ `T (m ≤ᵇ n)` □ `tt` □□□ □□□□□ `s≤s` □□□ `m≤n` □□□ `suc m ≤ᵇ suc n` □□□ `m ≤ᵇ n` □□□□□□□□□□□□□ □□□ `T (m ≤ᵇ n)` □□□□

□□□ □□□

□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□

□□□□□□□

□□□ □□
□□□□□□□□ `Dec A` □□□ `Dec` □□□□□□Decidable□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□ `Dec A` □□□□□□□□□ `yes x` □□□□□□ `x` □□ `A` □□□□□□□□□□□ `no ¬x` □□□□□□ `x` □□□ `A` □□□□□□□□□□□□□□□□ `¬x` □□□□□ `A` □□□□□□□□□□□□□□□□

□□□□□□□□□□□□□ `_≤?_` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
¬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
```

□□□□□□□□□ `¬ (suc m ≤ zero)` □□□□□□□□□□□□□□□□□□□□□□□□ `zero ≤ n` □□ `suc m ≤ suc n` □□□□□□□□□□□□□ `suc m ≤ zero` □ □□□□□□ `¬ (m ≤ n)` □□□ `¬m≤n` □□□ `¬ (suc m ≤ suc n)` □□□□□□□□ `suc m ≤ suc n` □□□□□□□ `s≤s m≤n` □□□□□□□□□□□□□□□□ □□□□ `¬m≤n m≤n` □□□□

□□□□□□□□□□□□□□□□□□□□□

```
_≤?_ : ∀ (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)
```

□ `_≤ᵇ_` □□□□□□□□□□□□□□□□□□□ `zero ≤ n` □□□□□□ `z≤n` □□□ □□□□□□□ `suc m ≤ zero` □□□□□□□ `¬s≤z` □□□ □□□□□□□□□□□□□□□□□ `m ≤? n` □□□□□□□□□□ `yes` □□□□□□□□□ `m ≤ n` □□□ `m≤n` □□□ `s≤s m≤n` □□□□ `suc m ≤ suc n` □□□□□ `no` □□□□□ □□□□ `¬ (m ≤ n)` □□□ `¬m≤n` □□□ `¬s≤s ¬m≤n` □□□□ `¬ (suc m ≤ suc n)` □□□□

□□□□ `_≤ᵇ_` □□□□□□□□□□□□□□□□ `≤ᵇ→≤` □ `≤→≤ᵇ` □□□□□□□□ □□□□□ `_≤?_` □□□□□□□□□□□□□□□□□□□□□□□□ `_≤?_` □□□□□ `_≤ᵇ_` □ `≤ᵇ→≤` □ `≤→≤ᵇ` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ `_≤?_` □□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
_ : 2 ≤? 4 ≡ yes (s≤s (s≤s z≤n))
_ = refl

_ : 4 ≤? 2 ≡ no (¬s≤s (¬s≤s ¬s≤z))
_ = refl
```

□□□□□ Agda □□□□□□□□□□□□ `C-c C-n` □□□ `2 ≤? 4` □□ `4 ≤? 2` □□ □□□□□□□Agda □□□□□□□□

□□□□□□□□□□□ `¬s≤z` □ `¬s≤s` □□□□□□□□□□□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□

□□ `_<?_` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□

```
postulate
  _<?_ : ∀ (m n : ℕ) → Dec (m < n)
```

```
-- □□□□□□□□□□
```

□□ `_≡ℕ?_` □□□□

□□□□□□□□□□□□□□□□□□□□

```
postulate
  _≡ℕ?_ : ∀ (m n : ℕ) → Dec (m ≡ n)
```

```
-- □□□□□□□□□□
```

□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□ `m ≤ᵇ n` □□□□□□□□ `m ≤ n` □□□□□□ □□□□□□□□□□□□□□□□□□□□

```
_≤?′_ : ∀ (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
```

□□ `m ≤ᵇ n` □□□□□ `≤ᵇ→≤` □□□□□ `m ≤ n` □□□□□□ □□ `m ≤ᵇ n` □□□□□ `≤→≤ᵇ` □□□□ `m ≤ n` □□□□□□□□□□□□□□□□

□□□□□□□`with` □□□□□□□□□□□□□□□□□□□□□□□

```
_≤?″_ : ∀ (m n : ℕ) → Dec (m ≤ n)
m ≤?″ n with m ≤ᵇ n
... | true   =  yes (≤ᵇ→≤ m n tt)
... | false  =  no (≤→≤ᵇ {m} {n})
```

□□ Agda □□□□□□□□□□□□□

```
⊤ !=< (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
```

□□□□□ with □□□□□ Agda □□□□□□ m ≤ᵇ n □□□ T (m ≤ᵇ n) □ ⊤ □□ m ≤ᵇ n □□□ T (m ≤ᵇ n) □ ⊥ □

□□□□□□□□□□ _≤?_ □□□□□□□□□□□□□□□□□□ _≤ᵇ_ □ □□□□□□□□□□□ _≤?_ □□□□□□□□□□□□□□□

□□□Erasure□□□□□□□□□□□□□□

```
⌊_⌋ : ∀ {A : Set} → Dec A → Bool
⌊ yes x ⌋ = true
⌊ no ¬x ⌋ = false
```

□□□□□□□□□□ _≤?_ □□□ _≤ᵇ_ □

```
_≤ᵇ′_ : ℕ → ℕ → Bool
m ≤ᵇ′ n = ⌊ m ≤? n ⌋
```

□□□□□□□ D □□□□□ Dec A □□□□ T ⌊ D ⌋ □□□ A □□□□□

```
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
```

□□□□□□□□□□□□ T (m ≤ᵇ′ n) □□□ m ≤ n □□□□□

```
≤ᵇ′→≤ : ∀ {m n : ℕ} → T (m ≤ᵇ′ n) → m ≤ n
≤ᵇ′→≤ = toWitness

≤→≤ᵇ′ : ∀ {m n : ℕ} → m ≤ n → T (m ≤ᵇ′ n)
≤→≤ᵇ′ = fromWitness
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□

```
infixr 6 _∧_

_∧_ : Bool → Bool → Bool
true ∧ true = true
false ∧ _    = false
_     ∧ false = false
```

□ Emacs □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□

```
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 }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ Emacs □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□

```
infixr 5 _∨_

_∨_ : Bool → Bool → Bool
true ∨ _      = true
_    ∨ true   = true
false ∨ false = false
```

□ Emacs □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□

```
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 }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ Emacs □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□

□□□□□□□□□□□□□□□□

```
not : Bool → Bool
not true = false
not false = true
```

□□□□□□□□□□□□□□□□□□□

```
¬? : ∀ {A : Set} → Dec A → Dec (¬ A)
¬? (yes x) = no (¬¬-intro x)
¬? (no ¬x) = yes ¬x
```

□□□□ yes □ no □□□□□□□□□□□□ `¬ A` □□□□□□□ `¬ ¬ A` □□——□□□□ `A` □□□□□□□□□□

□□□□□□□□□□□□□□□□□□

```
_⊃_ : Bool → Bool → Bool
_     ⊃ true   = true
false ⊃ _      = true
true ⊃ false = false
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□ Emacs □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□

```
_→-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))
```

[illegible] `f` [illegible] `x` [illegible] `¬y` [illegible]

[illegible] Emacs [illegible]

[illegible] `erasure` [illegible]

[illegible]

```
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 ⌋
```

[illegible] `iff-erasure` [illegible]

[illegible] `_↔_` [illegible]

```
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 ⌋
```

```
-- [illegible]
```

[illegible]

[illegible] `monus` [illegible] *guarded* [illegible]——[illegible] `n ≤ m` [illegible] `m` [illegible] `n` [illegible]

```
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
```

[illegible] `n ≤ m` [illegible]

```
_ : minus 5 3 (s≤s (s≤s (s≤s z≤n))) ≡ 2
_ = refl
```

[illegible] *proof by reflection* [illegible] Agda [illegible] `n ≤? m` [illegible] `n ≤ m` [illegible]

[illegible] Agda [illegible] Agda [illegible] Agda [illegible] `⊤` [illegible]

□□□□□□□□□□□□□□ `T ⌊ n ≤? m ⌋` □□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□ `n ≤? m` □□□□□□□□ `n ≤ m` □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ `T` □□□□□□ `T` □□□□□□□□□□□□□□ `true` □□□□□□□□ `⊤` □ `false` □□□□□□ `⊥` □□□□□□□□□□□□□□□□□□□□□□□□□□

- □□ `n ≤ m` □□□□□□□□□□□□□□ `⊤` □ □□ Agda □□□□□□□□□□□□
- □□□□□□□□□ `⊥` □Agda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `3 - 5` □□□ `_n≤m_254 : ⊥` □

□□□□□□□□□□ `toWitness` □□□ `n ≤ m` □□□□

```
_-_ : (m n : ℕ) {n≤m : T ⌊ n ≤? m ⌋} → ℕ
_-_ m n {n≤m} = minus m n (toWitness n≤m)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□ `_-_` □□

```
_ : 5 - 3 ≡ 2
_ = refl
```

□□□□□□□□□□□□□□□□□□□□□ `T ⌊ ? ⌋` □□□□□ `True` □□□□□

```
True : ∀ {Q} → Dec Q → Set
True Q = T ⌊ Q ⌋
```

□□ `False`

□□ `True` □ `toWitness` □ `fromWitness` □□□□□□□□□□□ `False` □ `toWitnessFalse` □ `fromWitnessFalse` □

□□□

```
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  □□□ (\and, \wedge)
∨  U+2228  □□□ (\or, \vee)
⊃  U+2283  □□ (\sup)
ᵇ  U+1D47  □□□□□ B  (\^b)
⌊  U+230A  □□□□□ (\clL)
⌋  U+230B  □□□□□ (\clR)
```

Chapter 10

Lists: □□□□□□□

```
module plfa.part1.Lists where
```

□□□□□□□□List□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□Polymorphic Types□□□□□□□Higher-order Functions□□□□□

□□

```
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 (_≃_; _⇔_)
```

□□

Agda □□□□□□□□□□

```
data List (A : Set) : Set where
  [] : List A
  _∷_ : A → List A → List A

infixr 5 _∷_
```

□□□□□□□□□□□□□□ `A` □□□□□□□ `List A` □□□□□□□□□□□□□□□□□ `[]` □□□ *nil*□□□□□□□ `A` □□□□□□□□□□□□□□`_∷_`□□□ *cons*□□ *constructor* □□□□□□□□□□ `A` □□□□□□□□□ `List A` □□□□□□□□□□ `List A` □□□`_∷_` □□□□□□□□ 5□□□□□□

[illegible]

```
_ : List ℕ
_ = 0 ∷ 1 ∷ 2 ∷ []
```

[illegible] `_∷_` [illegible] `0 ∷ (1 ∷ (2 ∷ []))` [illegible] `0` [illegible]Head[illegible] `1 ∷ (2 ∷ [])` [illegible] Tail[illegible]

[illegible]

```
data List′ : Set → Set where
  []′ : ∀ {A : Set} → List′ A
  _∷′_ : ∀ {A : Set} → A → List′ A → List′ A
```

[illegible]

```
_ : List ℕ
_ = _∷_ {ℕ} 0 (_∷_ {ℕ} 1 (_∷_ {ℕ} 2 ([] {ℕ})))
```

[illegible]

[illegible]

```
{-# BUILTIN LIST List #-}
```

[illegible] Agda[illegible] `List` [illegible] Haskell [illegible] `[]` [illegible] `_∷_` [illegible] nil [illegible] cons[illegible]

[illegible]

[illegible]

```
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 ∷ []
```

[illegible] `[ x , y , z ]` [illegible] `x ∷ y ∷ z ∷ []` [illegible]

[illegible]

[illegible] `_++_` [illegible]Append[illegible]

```
infixr 5 _++_

_++_ : ∀ {A : Set} → List A → List A → List A
[] ++ ys       = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)
```

`A` [illegible]**Polymorphic**[illegible] [illegible]
[illegible]

□□□□□□□□□□□□□□□□□□□□

```
_ : [ 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 ∷ []
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
++-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)
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□ `[]` □□□□□□□□□ □□□□□□□□□□ `x ∷ xs` □□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□ `++-assoc xs ys zs` □

□□□ Agda □□□□□□□ `cong (x ∷_)` □□□□□□□

```
(xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
```

□□□□□

```
x ∷ ((xs ++ ys) ++ zs) ≡ x ∷ (xs ++ (ys ++ zs))
```

□□□□□□□

□□□□□□□□□□ `[]` □ `_++_` □□□□□□□□□ □□□□□□□□□□□□□□

```
++-identityˡ : ∀ {A : Set} (xs : List A) → [] ++ xs ≡ xs
++-identityˡ xs =
  begin
    [] ++ xs
  ≡⟨⟩
    xs
  ∎
```

□□□□□□□□□□□□□□□□

```
++-identityʳ : ∀ {A : Set} (xs : List A) → xs ++ [] ≡ xs
++-identityʳ [] =
  begin
    [] ++ []
  ≡⟨⟩
    []
  ∎
++-identityʳ (x ∷ xs) =
  begin
    (x ∷ xs) ++ []
  ≡⟨⟩
    x ∷ (xs ++ [])
  ≡⟨ cong (x ∷_) (++-identityʳ xs) ⟩
    x ∷ xs
  ∎
```

□□□□□□□□□□□□□□□□□ `_++_` □ `[]` □□□□□□□□□□□□□Monoid□□

□□

□□□□□□□□□□□□□□□□□□□

```
length : ∀ {A : Set} → List A → ℕ
length []       = zero
length (x ∷ xs) = suc (length xs)
```

□□□□□□□□□□□ `A` □ □□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□

```
_ : 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))
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□ `length []` □□□□□ `length {ℕ} []` □ □□ `[]` □□□□Agda □□□□□□□□□□□□□□□□

□□□□

□□□□□□□□□□□□□□□□□□□□□□

```
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
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□□□□ `[]` □□□□□□□□□□ □□□□□Agda □□□□ `length` □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□ `x ∷ xs` □□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□ `length-++ xs ys` □ □ `cong suc` □□□□

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
reverse : ∀ {A : Set} → List A → List A
reverse []       = []
reverse (x ∷ xs) = reverse xs ++ [ x ]
```

□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□

```
_ : 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 ]
∎
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `n` □□□□□ □□□□ `1`□`2` □□ `n - 1` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ `n * (n - 1) / 2` □□□□□□□□□□□□□□□□□□□□□□

□□ `reverse-++-distrib` □□□□

□□□

```
reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
```

□□ `reverse-involutive` □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Involution□□ □□□□□□□□□□

```
reverse (reverse xs) ≡ xs
```

□□□□

□□ □□□□□□□□□□□□□□□□□□□□□□

```
shunt : ∀ {A : Set} → List A → List A → List A
shunt []       ys = ys
shunt (x ∷ xs) ys = shunt xs (x ∷ ys)
```

□□□□□□□□□□□□□□□□□□□□□□□_□_□□□□□□□□□□□□□□□□□□□ □□_□_□

□□□Shunt□□□□□□□□□□

```
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
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□□□□ `[]` □□□□□□□□□□ □□□□□□□□□□□ `x ∷ xs` □□□□□□□□□□□□□□□□ □□□

□□ □□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
reverse′ : ∀ {A : Set} → List A → List A
reverse′ xs = shunt xs []
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
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
  ∎
```

□□□□□□□□□□□□□□□□ `[ 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 ∷ []
  ∎
```

□□□□□□□□□□□□□□□□□□□□□□□□

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□Higher-Order Function□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
map : ∀ {A B : Set} → (A → B) → List A → List B
map f []        = []
map f (x ∷ xs) = f x ∷ map f xs
```

□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
_ : 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 ∷ []
  ∎
```

□□□□□□□□□□□□□□□□

□□□

```
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 ]
  ∎
```

[illegible] [illegible]
n [illegible] [illegible] *n* [illegible]

[illegible] `map-compose` [illegible]

[illegible]

```
map (g ∘ f) ≡ map g ∘ map f
```

[illegible]

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

[illegible] `map-++-distribute` [illegible]

[illegible]

```
map f (xs ++ ys) ≡ map f xs ++ map f ys
```

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

[illegible] `map-Tree` [illegible]

[illegible] `A` [illegible] `B` [illegible]

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  node : Tree A B → B → Tree A B → Tree A B
```

[illegible]

```
map-Tree : ∀ {A B C D : Set} → (A → C) → (B → D) → Tree A B → Tree C D
```

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

[illegible]

[illegible] [illegible]

```
foldr : ∀ {A B : Set} → (A → B → B) → B → List A → B
foldr _⊗_ e []       = e
foldr _⊗_ e (x :: xs) = x ⊗ foldr _⊗_ e xs
```

[illegible] [illegible]

[illegible]

```
_ : 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)))
  ∎
```

[illegible]

[illegible] [illegible]

```
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
  ∎
```

[illegible] `[]` [illegible] `_∷_` [illegible] `e` [illegible] `_⊗_` [illegible] *n* [illegible] *n* [illegible]

[illegible]

```
foldr _∷_ [] xs ≡ xs
```

[illegible] `xs` [illegible] `List A` [illegible] `foldr` [illegible] `A` [illegible] `A` [illegible] `B` [illegible] `List A` [illegible]

```
xs ++ ys ≡ foldr _∷_ ys xs
```

[illegible]

[illegible] `product` [illegible]

[illegible]

```
product [ 1 , 2 , 3 , 4 ] ≡ 24
```

```
-- □□□□□□□□
```

□□ `foldr-++` □□□□

□□□□□□□□□□□□□□

```
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
```

□□ `map-is-foldr`

□□□□□□□□□□□□

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

□□□□□□□□□

□□ `map-is-foldr` □□□□

□□□ map □□□ fold □□□□

```
map f ≡ foldr (λ x xs → f x ∷ xs) []
```

□□□□□□□□□

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

□□ `fold-Tree` □□□□

□□□□□□□□□□□□□□□□□□□□□□□

```
fold-Tree : ∀ {A B C : Set} → (A → C) → (C → B → C → C) → Tree A B → C
```

```
-- □□□□□□□□□
```

□□ `map-is-fold-Tree` □□□□

□□□□□□□□□□□□ `map-is-foldr` □□□□□□

```
-- □□□□□□□□□
```

□□ `sum-downFrom` □□□□

□□□□□□□□□□□□□

```
downFrom : ℕ → List ℕ
downFrom zero    = []
downFrom (suc n) = n ∷ downFrom n
```

□□□

```
_ : downFrom 3 ≡ [ 2 , 1 , 0 ]
_ = refl
```

□□□□□□ `(n - 1) + ⋯ + 0` □□ `n * (n ∸ 1) / 2`□

```
sum (downFrom n) * 2 ≡ n * (n ∸ 1)
```

□□□

□□ □□□□□□□□□□□□□□□□□□□□□□**Monoid**□□

□□□□□□□□□□□□□□□□□□□□□

```
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
```

□□□□□□□□□□□□□□□□□□□□□

```
+-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
    }
```

□□ `_⊗_` □ `e` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
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
  ∎
```

□□□□□□□□□□□□□□□□□□

```
postulate
  foldr-++ : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) (xs ys : List A) →
    foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ (foldr _⊗_ e ys) xs
```

□□□□□□□□□□□□□□□□□□□□□□□□

```
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
  ∎
```

□□ `foldl` □□□□

□□□□□□ `foldl` □□ `foldr` □□□□□□□□□□□□□□□□□□□□□□

```
foldr _⊗_ e [ x , y , z ]  =  x ⊗ (y ⊗ (z ⊗ e))
foldl _⊗_ e [ x , y , z ]  =  ((e ⊗ x) ⊗ y) ⊗ z
```

```
-- □□□□□□□□□
```

□□ `foldr-monoid-foldl` □□□□

□□□□ `_⊗_` □ `e` □□□□□□□□ `foldr _⊗_ e` □ `foldl _⊗_ e` □□□ □□□□□□□

```
-- □□□□□□□□□
```

□□

□□□□□□□□□□□□□□□□□□□□□□□□ `All` □ `Any` □

□□ `All P` □□□□□□□□□□□ `P` □□□□

```
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)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `P` □□□□□□□□□□□□□ □□□□□□□□ `P` □□□□□□□□□□□□□□□□□□□□□□ `P` □□□□□□□□□□□□□□ Agda □□□□□□□□□□□□□□□□□□□□□□□□ `All P` □□□□□□

□□□□ `All (_≤ 2)` □□□□□□□□□□□□□□□□□□□□□ □□ `z≤n` □□□□□□□ `n` □ `zero ≤ n` □□□ □□□□ `m` □ `n` □□□ `m≤n` □□□ `m ≤ n` □□□ `s≤s m≤n` □□□ `suc m ≤ suc n`:

```
_ : All (_≤ 2) [ 0 , 1 , 2 ]
_ = z≤n ∷ s≤s z≤n ∷ s≤s (s≤s z≤n) ∷ []
```

□□ `_∷_` □ `[]` □ `All P` □□□□□□□□ `List A` □□ □□□□□□□ `0 ≤ 2` □ `1 ≤ 2` □ `2 ≤ 2` □□□□

□□□□□□□□□□ `[_,_,_]` □□□□□□□□□□□ `All` □□□□ □□□ `List` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□ `List` □□ `[_,_,_]` □□□□ `All` □ □□□□□□

[illegible]

[illegible] `Any P` [illegible] `P` [illegible]

```
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)
```

[illegible] `P` [illegible] `P` [illegible]

```
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)
```

[illegible] `[ 0 , 1 , 0 , 2 ]` [illegible]

```
_ : 0 ∈ [ 0 , 1 , 0 , 2 ]
_ = here refl

_ : 0 ∈ [ 0 , 1 , 0 , 2 ]
_ = there (there (here refl))
```

[illegible]

```
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 ()))))
```

`()` [illegible] `3 ≡ 0` [illegible] `3 ≡ 1` [illegible] `3 ≡ 0` [illegible] `3 ≡ 2` [illegible] `3 ∈ []` [illegible]

[illegible]

[illegible]

```
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 ⟩
```

□□ `Any-++-⇔` □□□□

□□ `Any` □□ `All` □□□□□□ `_×_` □□□□□□□□□□□ `All-++-⇔` □□□□ □□□□□□□□□ `_∈_` □ `_++_` □□□□□□□□

```
-- □□□□□□□□□
```

□□ `All-++-≃` □□□□

□□ `All-++-⇔` □□□□□□□□□□□□□□□□□□□

```
-- □□□□□□□□□
```

□□ `¬Any⇔All¬` □□□□

□□□ `Any` □ `All` □□□□□□□□□□□□□□

```
(¬_ ∘ Any P) xs ⇔ All (¬_ ∘ P) xs
```

□□□□□□□□□□□□ `_∘_` □□□□□□□□□□□□□□ □□□□□□□□□□

□□□□□□□□□□□

```
(¬_ ∘ All P) xs ⇔ Any (¬_ ∘ P) xs
```

□□□□□□□□□□□□□□□□□

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

□□ `¬Any≃All¬` □□□□

□□□□□□ `¬Any⇔All¬` □□□□□□□□□□□□ □□□□□□□□□□

```
-- □□□□□□□□
```

□□ `All-∀` □□□□

□□□ `All P xs` □□□ `∀ x → x ∈ xs → P x`.

```
-- □□□□□□□□□
```

□□ `Any-∃` □□□□

□□□ `Any P xs` □□□ `∃[ x ] (x ∈ xs × P x)`.

```
-- □□□□□□□□□
```

□□□□□□□□□□□□□□□□□□□□□□

□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `All` □□□□□□□□□□□□□□□□□□□

```
all : ∀ {A : Set} → (A → Bool) → List A → Bool
all p = foldr _∧_ true ∘ map p
```

□□□□□□□□ `map` □ `foldr` □□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□ `All` □□□□□□□□ `P` □□□□□ `A → Set` □□□□□□□□□□□ `A` □□ `x` □□□ `P x` □ `x` □□ □□□□□□ `P` □□□□□Decidable□□□□□□□□□□□□□□□□ `x` □□□□□ `P x`□

```
Decidable : ∀ {A : Set} → (A → Set) → Set
Decidable {A} P = ∀ (x : A) → Dec (P x)
```

□□□□ `P` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
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 }
```

□□□□□□□ `P` □□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□ `_∷_` □□□ `⟨_,_⟩` □□□□□□□□□□□□

□□ `Any?` □□□□

□□ `All` □□□□ `all` □ `All?` □□□□□□□□□□□□□□□□□□□□□□□ □□ `Any` □□□□□ `any` □ `Any?` □□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□

```
-- □□□□□□□□□□
```

□□ `split` □□□□

□□ `merge` □□□□□□□□□□□□□□□□□□□□□□□

```
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)
```

□□

```
_ : merge [ 1 , 4 ] [ 2 , 3 ] [ 1 , 2 , 3 , 4 ]
_ = left-∷ (right-∷ (right-∷ (left-∷ [])))
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□ `filter` □□□□□□□□□□□□□□□□□□□□□ □□ □□□□□□□□□□□

```
split : ∀ {A : Set} {P : A → Set} (P? : Decidable P) (zs : List A)
  → ∃[ xs ] ∃[ ys ] ( merge xs ys zs × All P xs × All (¬_ ∘ P) ys )
```

```
-- □□□□□□□□
```

□□□

□□□□□□□□□□□□□□□□□□□

```
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)
```

□□□□□ `IsMonoid` □□□□□□□□□□□□□□□□□□□□□□□□□□□

`Relation.Unary` [illegible] `Relation.Binary` [illegible] `Decidable` [illegible] [illegible] `P` [illegible] `_≤_` [illegible]

Unicode

[illegible] Unicode[illegible]

```
∷  U+2237  [illegible]  (\::)
⊗  U+2297  [illegible]  (\otimes, \ox)
∈  U+2208  [illegible]  (\in)
∉  U+2209  [illegible]  (\inn, \notin)
```

Part II

[illegible]

Chapter 11

Lambda: Introduction to Lambda Calculus

```
module plfa.part2.Lambda where
```

The *lambda-calculus*, first published by the logician Alonzo Church in 1932, is a core calculus with only three syntactic constructs: variables, abstraction, and application. It captures the key concept of *functional abstraction*, which appears in pretty much every programming language, in the form of either functions, procedures, or methods. The *simply-typed lambda calculus* (or STLC) is a variant of the lambda calculus published by Church in 1940. It has the three constructs above for function types, plus whatever else is required for base types. Church had a minimal base type with no operations. We will instead echo Plotkin's *Programmable Computable Functions* (PCF), and add operations on natural numbers and recursive function definitions.

This chapter formalises the simply-typed lambda calculus, giving its syntax, small-step semantics, and typing rules. The next chapter Properties proves its main properties, including progress and preservation. Following chapters will look at a number of variants of lambda calculus.

Be aware that the approach we take here is *not* our recommended approach to formalisation. Using de Bruijn indices and intrinsically-typed terms, as we will do in Chapter DeBruijn, leads to a more compact formulation. Nonetheless, we begin with named variables and extrinsically-typed terms, partly because names are easier than indices to read, and partly because the development is more traditional.

The development in this chapter was inspired by the corresponding development in Chapter *Stlc* of *Software Foundations* (*Programming Language Foundations*). We differ by representing contexts explicitly (as lists pairing identifiers with types) rather than as partial maps (which take identifiers to types), which corresponds better to our subsequent development of DeBruijn notation. We also differ by taking natural numbers as the base type rather than booleans, allowing more sophisticated examples. In particular, we will be able to show (twice!) that two plus two is four.

Imports

```
open import Data.Bool using (T; not)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (List; _∷_; [])
open import Data.Nat using (ℕ; zero; suc)
open import Data.Product using (∃-syntax; _×_)
open import Data.String using (String; _≟_)
open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Relation.Nullary.Decidable using (⌊_⌋; False; toWitnessFalse)
open import Relation.Nullary.Negation using (¬?)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl)
```

Syntax of terms

Terms have seven constructs. Three are for the core lambda calculus:

- Variables `` ` x ``
- Abstractions `ƛ x ⇒ N`
- Applications `L · M`

Three are for the naturals:

- Zero `` `zero ``
- Successor `` `suc M ``
- Case `case L [zero⇒ M |suc x ⇒ N ]`

And one is for recursion:

- Fixpoint `μ x ⇒ M`

Abstraction is also called *lambda abstraction*, and is the construct from which the calculus takes its name.

With the exception of variables and fixpoints, each term form either constructs a value of a given type (abstractions yield functions, zero and successor yield natural numbers) or deconstructs it (applications use functions, case terms use naturals). We will see this again when we come to the rules for assigning types to terms, where constructors correspond to introduction rules and deconstructors to eliminators.

Here is the syntax of terms in Backus-Naur Form (BNF):

```
L, M, N  ::=
  ` x  |  ƛ x ⇒ N  |  L · M  |
  `zero  |  `suc M  |  case L [zero⇒ M |suc x ⇒ N ]  |
  μ x ⇒ M
```

And here it is formalised in Agda:

```
Id : Set
Id = String

infix  5  ƛ_⇒_
infix  5  μ_⇒_
infixl 7  _·_
infix  8  `suc_
infix  9  `_

data Term : Set where
```

```
  `_                      : Id → Term
  ƛ_⇒_                    : Id → Term → Term
  _·_                     : Term → Term → Term
  `zero                   : Term
  `suc_                   : Term → Term
  case_[zero⇒_|suc_⇒_]    : Term → Term → Id → Term → Term
  μ_⇒_                    : Id → Term → Term
```

We represent identifiers by strings. We choose precedence so that lambda abstraction and fixpoint bind least tightly, then application, then successor, and tightest of all is the constructor for variables. Case expressions are self-bracketing.

Example terms

Here are some example terms: the natural number two, a function that adds naturals, and a term that computes two plus two:

```
two : Term
two = `suc `suc `zero

plus : Term
plus = μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
         case ` "m"
           [zero⇒ ` "n"
           |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

The recursive definition of addition is similar to our original definition of `_+_` for naturals, as given in Chapter Naturals. Here variable “m” is bound twice, once in a lambda abstraction and once in the successor branch of the case; the first use of “m” refers to the former and the second to the latter. Any use of “m” in the successor branch must refer to the latter binding, and so we say that the latter binding *shadows* the former. Later we will confirm that two plus two is four, in other words that the term

```
plus · two · two
```

reduces to `` `suc `suc `suc `suc `zero ``.

As a second example, we use higher-order functions to represent natural numbers. In particular, the number *n* is represented by a function that accepts two arguments and applies the first *n* times to the second. This is called the *Church representation* of the naturals. Here are some example terms: the Church numeral two, a function that adds Church numerals, a function to compute successor, and a term that computes two plus two:

```
twoᶜ : Term
twoᶜ = ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")

plusᶜ : Term
plusᶜ = ƛ "m" ⇒ ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒
         ` "m" · ` "s" · (` "n" · ` "s" · ` "z")

sucᶜ : Term
sucᶜ = ƛ "n" ⇒ `suc (` "n")
```

The Church numeral for two takes two arguments `s` and `z` and applies `s` twice to `z`. Addition takes two numerals `m` and `n`, a function `s` and an argument `z`, and it uses `m` to apply `s` to the result of using `n` to apply `s` to `z`; hence `s` is applied `m` plus `n` times to `z`, yielding the Church numeral for the sum of `m` and `n`. For convenience, we define a function that computes successor. To convert a Church numeral to the corresponding natural, we apply it to the `sucᶜ` function and the natural number zero. Again, later we will confirm that two plus two is four, in other words that the term

```
plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

reduces to `` `suc `suc `suc `suc `zero ``.

Exercise `mul` (recommended)

Write out the definition of a lambda term that multiplies two natural numbers. Your definition may use `plus` as defined earlier.

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

Exercise `mulᶜ` (practice)

Write out the definition of a lambda term that multiplies two natural numbers represented as Church numerals. Your definition may use `plusᶜ` as defined earlier (or may not — there are nice definitions both ways).

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

Exercise `primed` (stretch)

Some people find it annoying to write `` ` "x" `` instead of `x`. We can make examples with lambda terms slightly easier to write by adding the following definitions:

```
ƛ′_⇒_ : Term → Term → Term
ƛ′ (` x) ⇒ N = ƛ x ⇒ N
ƛ′ _ ⇒ _     = ⊥-elim impossible
  where postulate impossible : ⊥

case′_[zero⇒_|suc_⇒_] : Term → Term → Term → Term → Term
case′ L [zero⇒ M |suc (` x) ⇒ N ] = case L [zero⇒ M |suc x ⇒ N ]
case′ _ [zero⇒ _ |suc _ ⇒ _ ]     = ⊥-elim impossible
  where postulate impossible : ⊥

μ′_⇒_ : Term → Term → Term
μ′ (` x) ⇒ N = μ x ⇒ N
μ′ _ ⇒ _     = ⊥-elim impossible
  where postulate impossible : ⊥
```

We intend to apply the function only when the first term is a variable, which we indicate by postulating a term `impossible` of the empty type `⊥`. If we use C-c C-n to normalise the term

```
ƛ′ two ⇒ two
```

Agda will return an answer warning us that the impossible has occurred:

```
⊥-elim (plfa.part2.Lambda.impossible (`` `suc (`suc `zero)) (`suc (`suc `zero)) ``)
```

While postulating the impossible is a useful technique, it must be used with care, since such postulation could allow us to provide evidence of *any* proposition whatsoever, regardless of its truth.

The definition of `plus` can now be written as follows:

```
plus′ : Term
plus′ = μ′ + ⇒ ƛ′ m ⇒ ƛ′ n ⇒
          case′ m
            [zero⇒ n
            |suc m ⇒ `suc (+ · m · n) ]
  where
  + = ` "+"
  m = ` "m"
  n = ` "n"
```

Write out the definition of multiplication in the same style.

Formal vs informal

In informal presentation of formal semantics, one uses choice of variable name to disambiguate and writes `x` rather than `` ` x `` for a term that is a variable. Agda requires we distinguish.

Similarly, informal presentation often use the same notation for function types, lambda abstraction, and function application in both the *object language* (the language one is describing) and the *meta-language* (the language in which the description is written), trusting readers can use context to distinguish the two. Agda is not quite so forgiving, so here we use `ƛ x ⇒ N` and `L · M` for the object language, as compared to `λ x → N` and `L M` in our meta-language, Agda.

Bound and free variables

In an abstraction `ƛ x ⇒ N` we call `x` the *bound* variable and `N` the *body* of the abstraction. A central feature of lambda calculus is that consistent renaming of bound variables leaves the meaning of a term unchanged. Thus the five terms

- `` ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") ``
- `` ƛ "f" ⇒ ƛ "x" ⇒ ` "f" · (` "f" · ` "x") ``
- `` ƛ "sam" ⇒ ƛ "zelda" ⇒ ` "sam" · (` "sam" · ` "zelda") ``
- `` ƛ "z" ⇒ ƛ "s" ⇒ ` "z" · (` "z" · ` "s") ``
- `` ƛ "😇" ⇒ ƛ "😈" ⇒ ` "😇" · (` "😇" · ` "😈" ) ``

are all considered equivalent. Following the convention introduced by Haskell Curry, who used the Greek letter `α` (*alpha*) to label such rules, this equivalence relation is called *alpha renaming*.

As we descend from a term into its subterms, variables that are bound may become free. Consider the following terms:

- `` ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") `` has both `s` and `z` as bound variables.
- `` ƛ "z" ⇒ ` "s" · (` "s" · ` "z") `` has `z` bound and `s` free.

- `` ` "s" · (` "s" · ` "z") `` has both `s` and `z` as free variables.

We say that a term with no free variables is *closed*; otherwise it is *open*. Of the three terms above, the first is closed and the other two are open. We will focus on reduction of closed terms.

Different occurrences of a variable may be bound and free. In the term

```
(ƛ "x" ⇒ ` "x") · ` "x"
```

the inner occurrence of `x` is bound while the outer occurrence is free. By alpha renaming, the term above is equivalent to

```
(ƛ "y" ⇒ ` "y") · ` "x"
```

in which `y` is bound and `x` is free. A common convention, called the *Barendregt convention*, is to use alpha renaming to ensure that the bound variables in a term are distinct from the free variables, which can avoid confusions that may arise if bound and free variables have the same names.

Case and recursion also introduce bound variables, which are also subject to alpha renaming. In the term

```
μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
  case ` "m"
    [zero⇒ ` "n"
    |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

notice that there are two binding occurrences of `m`, one in the first line and one in the last line. It is equivalent to the following term,

```
μ "plus" ⇒ ƛ "x" ⇒ ƛ "y" ⇒
  case ` "x"
    [zero⇒ ` "y"
    |suc "x′" ⇒ `suc (` "plus" · ` "x′" · ` "y") ]
```

where the two binding occurrences corresponding to `m` now have distinct names, `x` and `x′`.

Values

A *value* is a term that corresponds to an answer. Thus, `` `suc `suc `suc `suc `zero `` is a value, while `plus · two · two` is not. Following convention, we treat all function abstractions as values; thus, `plus` by itself is considered a value.

The predicate `Value M` holds if term `M` is a value:

```
data Value : Term → Set where

  V-ƛ : ∀ {x N}
      ---------------
    → Value (ƛ x ⇒ N)

  V-zero :
      -----------
      Value `zero
  V-suc : ∀ {V}
    → Value V
      --------------
    → Value (`suc V)
```

In what follows, we let `V` and `W` range over values.

Formal vs informal

In informal presentations of formal semantics, using `V` as the name of a metavariable is sufficient to indicate that it is a value. In Agda, we must explicitly invoke the `Value` predicate.

Other approaches

An alternative is not to focus on closed terms, to treat variables as values, and to treat `ƛ x ⇒ N` as a value only if `N` is a value. Indeed, this is how Agda normalises terms. We consider this approach in Chapter Untyped.

Substitution

The heart of lambda calculus is the operation of substituting one term for a variable in another term. Substitution plays a key role in defining the operational semantics of function application. For instance, we have

```
  (ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) · sucᶜ · `zero
⟶
  (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero
⟶
  sucᶜ · (sucᶜ · `zero)
```

where we substitute `sucᶜ` for `` ` "s" `` and `` `zero `` for `` ` "z" `` in the body of the function abstraction.

We write substitution as `N [ x := V ]`, meaning "substitute term `V` for free occurrences of variable `x` in term `N`", or, more compactly, "substitute `V` for `x` in `N`", or equivalently, "in `N` replace `x` by `V`". Substitution works if `V` is any closed term; it need not be a value, but we use `V` since in fact we usually substitute values.

Here are some examples:

- `` (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) [ "s" := sucᶜ ] `` yields `` ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z") ``.
- `` (sucᶜ · (sucᶜ · ` "z")) [ "z" := `zero ] `` yields `` sucᶜ · (sucᶜ · `zero) ``.
- `` (ƛ "x" ⇒ ` "y") [ "y" := `zero ] `` yields `` ƛ "x" ⇒ `zero ``.
- `` (ƛ "x" ⇒ ` "x") [ "x" := `zero ] `` yields `` ƛ "x" ⇒ ` "x" ``.
- `` (ƛ "y" ⇒ ` "y") [ "x" := `zero ] `` yields `` ƛ "y" ⇒ ` "y" ``.

In the last but one example, substituting `` `zero `` for `x` in `` ƛ "x" ⇒ ` "x" `` does *not* yield `` ƛ "x" ⇒ `zero ``, since `x` is bound in the lambda abstraction. The choice of bound names is irrelevant: both `` ƛ "x" ⇒ ` "x" `` and `` ƛ "y" ⇒ ` "y" `` stand for the identity function. One way to think of this is that `x` within the body of the abstraction stands for a *different* variable than `x` outside the abstraction, they just happen to have the same name.

We will give a definition of substitution that is only valid when term substituted for the variable is closed. This is because substitution by terms that are *not* closed may require renaming of bound variables. For example:

- `(ƛ "x" ⇒ ` "x" · ` "y") [ "y" := ` "x" · `zero]` should not yield `(ƛ "x" ⇒ ` "x" · (` "x" · `zero))`.

Instead, we should rename the bound variable to avoid capture:

- `(ƛ "x" ⇒ ` "x" · ` "y") [ "y" := ` "x" · `zero ]` should yield `ƛ "x′" ⇒ ` "x′" · (` "x" · `zero)`.

Here `x′` is a fresh variable distinct from `x`. Formal definition of substitution with suitable renaming is considerably more complex, so we avoid it by restricting to substitution by closed terms, which will be adequate for our purposes.

Here is the formal definition of substitution by closed terms in Agda:

```
infix 9 _[_:=_]

_[_:=_] : Term → Id → Term → Term
(` x) [ y := V ] with x ≟ y
... | yes _         = V
... | no _          = ` x
(ƛ x ⇒ N) [ y := V ] with x ≟ y
... | yes _         = ƛ x ⇒ N
... | no _          = ƛ x ⇒ N [ y := V ]
(L · M) [ y := V ]  = L [ y := V ] · M [ y := V ]
(`zero) [ y := V ]  = `zero
(`suc M) [ y := V ] = `suc M [ y := V ]
(case L [zero⇒ M |suc x ⇒ N ]) [ y := V ] with x ≟ y
... | yes _         = case L [ y := V ] [zero⇒ M [ y := V ] |suc x ⇒ N ]
... | no _          = case L [ y := V ] [zero⇒ M [ y := V ] |suc x ⇒ N [ y := V ] ]
(μ x ⇒ N) [ y := V ] with x ≟ y
... | yes _         = μ x ⇒ N
... | no _          = μ x ⇒ N [ y := V ]
```

Let’s unpack the first three cases:

- For variables, we compare `y`, the substituted variable, with `x`, the variable in the term. If they are the same, we yield `V`, otherwise we yield `x` unchanged.
- For abstractions, we compare `y`, the substituted variable, with `x`, the variable bound in the abstraction. If they are the same, we yield the abstraction unchanged, otherwise we substitute inside the body.
- For application, we recursively substitute in the function and the argument.

Case expressions and recursion also have bound variables that are treated similarly to those in lambda abstractions. Otherwise we simply push substitution recursively into the subterms.

Examples

Here is confirmation that the examples above are correct:

```
_ : (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) [ "s" := sucᶜ ] ≡ ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")
_ = refl

_ : (sucᶜ · (sucᶜ · ` "z")) [ "z" := `zero ] ≡ sucᶜ · (sucᶜ · `zero)
_ = refl

_ : (ƛ "x" ⇒ ` "y") [ "y" := `zero ] ≡ ƛ "x" ⇒ `zero
_ = refl

_ : (ƛ "x" ⇒ ` "x") [ "x" := `zero ] ≡ ƛ "x" ⇒ ` "x"
_ = refl

_ : (ƛ "y" ⇒ ` "y") [ "x" := `zero ] ≡ ƛ "y" ⇒ ` "y"
_ = refl
```

Quiz

What is the result of the following substitution?

```
(ƛ "y" ⇒ ` "x" · (ƛ "x" ⇒ ` "x")) [ "x" := `zero ]
```

1. `(ƛ "y" ⇒ ` "x" · (ƛ "x" ⇒ ` "x"))`
2. `(ƛ "y" ⇒ ` "x" · (ƛ "x" ⇒ `zero))`
3. `(ƛ "y" ⇒ `zero · (ƛ "x" ⇒ ` "x"))`
4. `(ƛ "y" ⇒ `zero · (ƛ "x" ⇒ `zero))`

Exercise `_[_:=_]′` (stretch)

The definition of substitution above has three clauses (`ƛ`, `case`, and `μ`) that invoke a `with` clause to deal with bound variables. Rewrite the definition to factor the common part of these three clauses into a single function, defined by mutual recursion with substitution.

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

Reduction

We give the reduction rules for call-by-value lambda calculus. To reduce an application, first we reduce the left-hand side until it becomes a value (which must be an abstraction); then we reduce the right-hand side until it becomes a value; and finally we substitute the argument for the variable in the abstraction.

In an informal presentation of the operational semantics, the rules for reduction of applications are written as follows:

```
L —→ L′
--------------- ξ-·₁
L · M —→ L′ · M

M —→ M′
--------------- ξ-·₂
V · M —→ V · M′

---------------------------- β-ƛ
(ƛ x ⇒ N) · V —→ N [ x := V ]
```

The Agda version of the rules below will be similar, except that universal quantifications are made explicit, and so are the predicates that indicate which terms are values.

The rules break into two sorts. Compatibility rules direct us to reduce some part of a term. We give them names starting with the Greek letter `ξ` (*xi*). Once a term is sufficiently reduced, it will consist of a constructor and a deconstructor, in our case `ƛ` and `·`, which reduces directly. We give them names starting with the Greek letter `β` (*beta*) and such rules are traditionally called *beta rules*.

A bit of terminology: A term that matches the left-hand side of a reduction rule is called a *redex*. In the redex `(ƛ x ⇒ N) · V`, we may refer to `x` as the *formal parameter* of the function, and `V` as the *actual parameter* of the function application. Beta reduction replaces the formal parameter by the actual parameter.

If a term is a value, then no reduction applies; conversely, if a reduction applies to a term then it is not a value. We will show in the next chapter that this exhausts the possibilities: every well-typed term either reduces or is a value.

For numbers, zero does not reduce and successor reduces the subterm. A case expression reduces its argument to a number, and then chooses the zero or successor branch as appropriate. A fixpoint replaces the bound variable by the entire fixpoint term; this is the one case where we substitute by a term that is not a value.

Here are the rules formalised in Agda:

```
infix 4 _—→_

data _—→_ : Term → Term → Set where

  ξ-·₁ : ∀ {L L′ M}
    → L —→ L′
      -----------------
    → L · M —→ L′ · M

  ξ-·₂ : ∀ {V M M′}
    → Value V
    → M —→ M′
      -----------------
    → V · M —→ V · M′

  β-ƛ : ∀ {x N V}
    → Value V
      ------------------------------
    → (ƛ x ⇒ N) · V —→ N [ x := V ]

  ξ-suc : ∀ {M M′}
    → M —→ M′
      ------------------
    → `suc M —→ `suc M′
  ξ-case : ∀ {x L L′ M N}
    → L —→ L′
      -----------------------------------------------------------------
    → case L [zero⇒ M |suc x ⇒ N ] —→ case L′ [zero⇒ M |suc x ⇒ N ]

  β-zero : ∀ {x M N}
      ----------------------------------------
    → case `zero [zero⇒ M |suc x ⇒ N ] —→ M

  β-suc : ∀ {x V M N}
    → Value V
      ---------------------------------------------------
    → case `suc V [zero⇒ M |suc x ⇒ N ] —→ N [ x := V ]

  β-μ : ∀ {x M}
      ------------------------------
    → μ x ⇒ M —→ M [ x := μ x ⇒ M ]
```

The reduction rules are carefully designed to ensure that subterms of a term are reduced to values before the whole term is reduced. This is referred to as *call-by-value* reduction.

Further, we have arranged that subterms are reduced in a left-to-right order. This means that reduction is *deterministic*: for any term, there is at most one other term to which it reduces. Put another way, our reduction relation `—→` is in fact a function.

This style of explaining the meaning of terms is called a *small-step operational semantics*. If `M —→ N`, we say that term `M` *reduces* to term `N`, or equivalently, term `M` *steps* to term `N`. Each compatibility rule has another reduction rule in its premise; so a step always consists of a beta rule, possibly adjusted by zero or more compatibility rules.

Quiz

What does the following term step to?

```
(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")  —→  ???
```

1. `(ƛ "x" ⇒ ` "x")`
2. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`
3. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`

What does the following term step to?

```
(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")  —→  ???
```

1. `(ƛ "x" ⇒ ` "x")`
2. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`
3. `(ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x") · (ƛ "x" ⇒ ` "x")`

What does the following term step to? (Where `twoᶜ` and `sucᶜ` are as defined above.)

```
twoᶜ · sucᶜ · `zero  —→  ???
```

1. `sucᶜ · (sucᶜ · `zero)`
2. `(ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero`
3. `` `zero ``

Reflexive and transitive closure

A single step is only part of the story. In general, we wish to repeatedly step a closed term until it reduces to a value. We do this by defining the reflexive and transitive closure `—↠` of the step relation `—→`.

We define reflexive and transitive closure as a sequence of zero or more steps of the underlying relation, along lines similar to that for reasoning about chains of equalities in Chapter Equality:

```
infix 2 _—↠_
infix 1 begin_
infixr 2 _—→⟨_⟩_
infix 3 _∎

data _—↠_ : Term → Term → Set where
  _∎ : ∀ M
      ---------
    → M —↠ M

  _—→⟨_⟩_ : ∀ L {M N}
    → L —→ M
    → M —↠ N
      ---------
    → L —↠ N

begin_ : ∀ {M N}
  → M —↠ N
    ------
  → M —↠ N
begin M—↠N = M—↠N
```

We can read this as follows:

- From term `M`, we can take no steps, giving a step of type `M —↠ M`. It is written `M ∎`.
- From term `L` we can take a single step of type `L —→ M` followed by zero or more steps of type `M —↠ N`, giving a step of type `L —↠ N`. It is written `L —→⟨ L—→M ⟩ M—↠N`, where `L—→M` and `M—↠N` are steps of the appropriate type.

The notation is chosen to allow us to lay out example reductions in an appealing way, as we will see in the next section.

An alternative is to define reflexive and transitive closure directly, as the smallest relation that includes `—→` and is also reflexive and transitive. We could do so as follows:

```
data _—↠′_ : Term → Term → Set where

  step′ : ∀ {M N}
    → M —→ N
      -------
    → M —↠′ N

  refl′ : ∀ {M}
```

```
      -------
    → M —↠′ M

  trans′ : ∀ {L M N}
    → L —↠′ M
    → M —↠′ N
      -------
    → L —↠′ N
```

The three constructors specify, respectively, that `—↠′` includes `—→` and is reflexive and transitive. A good exercise is to show that the two definitions are equivalent (indeed, one embeds in the other).

Exercise `—↠≲—↠′` (practice)

Show that the first notion of reflexive and transitive closure above embeds into the second. Why are they not isomorphic?

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

Confluence

One important property a reduction relation might satisfy is to be *confluent*. If term `L` reduces to two other terms, `M` and `N`, then both of these reduce to a common term `P`. It can be illustrated as follows:

```
           L
          / \
         /   \
        /     \
       M       N
        \     /
         \   /
          \ /
           P
```

Here `L`, `M`, `N` are universally quantified while `P` is existentially quantified. If each line stands for zero or more reduction steps, this is called confluence, while if the top two lines stand for a single reduction step and the bottom two stand for zero or more reduction steps it is called the diamond property. In symbols:

```
postulate
  confluence : ∀ {L M N}
    → ((L —↠ M) × (L —↠ N))
      --------------------
    → ∃[ P ] ((M —↠ P) × (N —↠ P))

  diamond : ∀ {L M N}
    → ((L —→ M) × (L —→ N))
      --------------------
    → ∃[ P ] ((M —↠ P) × (N —↠ P))
```

The reduction system studied in this chapter is deterministic. In symbols:

```
postulate
  deterministic : ∀ {L M N}
    → L —→ M
    → L —→ N
      ------
    → M ≡ N
```

It is easy to show that every deterministic relation satisfies the diamond and confluence properties. Hence, all the reduction systems studied in this text are trivially confluent.

Examples

We start with a simple example. The Church numeral two applied to the successor function and zero yields the natural number two:

```
_ : twoᶜ · sucᶜ · `zero —↠ `suc `suc `zero
_ =
  begin
    twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero
  —→⟨ β-ƛ V-zero ⟩
    sucᶜ · (sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    sucᶜ · `suc `zero
  —→⟨ β-ƛ (V-suc V-zero) ⟩
    `suc (`suc `zero)
  ∎
```

Here is a sample reduction demonstrating that two plus two is four:

```
_ : plus · two · two —↠ `suc `suc `suc `suc `zero
_ =
  begin
    plus · two · two
  —→⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
    (ƛ "m" ⇒ ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two · two
  —→⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ "n" ⇒
      case two [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    case two [zero⇒ two |suc "m" ⇒ `suc (plus · ` "m" · two) ]
  —→⟨ β-suc (V-suc V-zero) ⟩
    `suc (plus · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
    `suc ((ƛ "m" ⇒ ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
    `suc ((ƛ "n" ⇒
      case `suc `zero [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two)
  —→⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
    `suc (case `suc `zero [zero⇒ two |suc "m" ⇒ `suc (plus · ` "m" · two) ])
  —→⟨ ξ-suc (β-suc V-zero) ⟩
    `suc `suc (plus · `zero · two)
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
    `suc `suc ((ƛ "m" ⇒ ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · `zero · two)
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
    `suc `suc ((ƛ "n" ⇒
      case `zero [zero⇒ ` "n" |suc "m" ⇒ `suc (plus · ` "m" · ` "n") ])
        · two)
  —→⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
    `suc `suc (case `zero [zero⇒ two |suc "m" ⇒ `suc (plus · ` "m" · two) ])
  —→⟨ ξ-suc (ξ-suc β-zero) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

And here is a similar sample reduction for Church numerals:

```
_ : plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero —↠ `suc `suc `suc `suc `zero
_ =
  begin
    (ƛ "m" ⇒ ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒ ` "m" · ` "s" · (` "n" · ` "s" · ` "z"))
      · twoᶜ · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
    (ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒ twoᶜ · ` "s" · (` "n" · ` "s" · ` "z"))
      · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ "s" ⇒ ƛ "z" ⇒ twoᶜ · ` "s" · (twoᶜ · ` "s" · ` "z")) · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ twoᶜ · sucᶜ · (twoᶜ · sucᶜ · ` "z")) · `zero
  —→⟨ β-ƛ V-zero ⟩
    twoᶜ · sucᶜ · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · ((ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (sucᶜ · (sucᶜ · `zero))
  —→⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (sucᶜ · (`suc `zero))
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
    (ƛ "z" ⇒ sucᶜ · (sucᶜ · ` "z")) · (`suc `suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    sucᶜ · (sucᶜ · `suc `suc `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    sucᶜ · (`suc `suc `suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

In the next chapter, we will see how to compute such reduction sequences.

Exercise `plus-example` (practice)

Write out the reduction sequence demonstrating that one plus one is two.

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

Syntax of types

We have just two types:

- Functions, `A ⇒ B`
- Naturals, `` `ℕ ``

As before, to avoid overlap we use variants of the names used by Agda.

Here is the syntax of types in BNF:

```
A, B, C  ::=  A ⇒ B | `ℕ
```

And here it is formalised in Agda:

```
infixr 7 _⇒_

data Type : Set where
  _⇒_ : Type → Type → Type
  `ℕ : Type
```

Precedence

As in Agda, functions of two or more arguments are represented via currying. This is made more convenient by declaring `_⇒_` to associate to the right and `_·_` to associate to the left. Thus:

- `` (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ `` stands for `` ((`ℕ ⇒ `ℕ) ⇒ (`ℕ ⇒ `ℕ)) ``.
- `plus · two · two` stands for `(plus · two) · two`.

Quiz

- What is the type of the following term?

  `` ƛ "s" ⇒ ` "s" · (` "s"  · `zero) ``

  1. `` (`ℕ ⇒ `ℕ) ⇒ (`ℕ ⇒ `ℕ) ``
  2. `` (`ℕ ⇒ `ℕ) ⇒ `ℕ ``
  3. `` `ℕ ⇒ (`ℕ ⇒ `ℕ) ``
  4. `` `ℕ ⇒ `ℕ ⇒ `ℕ ``
  5. `` `ℕ ⇒ `ℕ ``
  6. `` `ℕ ``

  Give more than one answer if appropriate.

- What is the type of the following term?

  `` (ƛ "s" ⇒ ` "s" · (` "s"  · `zero)) · sucᶜ ``

1. `(`ℕ ⇒ `ℕ) ⇒ (`ℕ ⇒ `ℕ)`
2. `(`ℕ ⇒ `ℕ) ⇒ `ℕ`
3. ``ℕ ⇒ (`ℕ ⇒ `ℕ)``
4. ``ℕ ⇒ `ℕ ⇒ `ℕ``
5. ``ℕ ⇒ `ℕ``
6. ``ℕ``

Give more than one answer if appropriate.

Typing

Contexts

While reduction considers only closed terms, typing must consider terms with free variables. To type a term, we must first type its subterms, and in particular in the body of an abstraction its bound variable may appear free.

A *context* associates variables with types. We let `Γ` and `Δ` range over contexts. We write `∅` for the empty context, and `Γ , x ⦂ A` for the context that extends `Γ` by mapping variable `x` to type `A`. For example,

- ``∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ``

is the context that associates variable `"s"` with type ``ℕ ⇒ `ℕ``, and variable `"z"` with type `` `ℕ ``.

Contexts are formalised as follows:

```
infixl 5 _,_⦂_

data Context : Set where
  ∅ : Context
  _,_⦂_ : Context → Id → Type → Context
```

Exercise `Context-≃` (practice)

Show that `Context` is isomorphic to `List (Id × Type)`. For instance, the isomorphism relates the context

```
∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ
```

to the list

```
[ ⟨ "z" , `ℕ ⟩ , ⟨ "s" , `ℕ ⇒ `ℕ ⟩ ]
```

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

Lookup judgment

We have two forms of *judgment*. The first is written

```
Γ ∋ x ⦂ A
```

and indicates in context `Γ` that variable `x` has type `A`. It is called *lookup*. For example,

- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "s" ⦂ `ℕ ⇒ `ℕ`

give us the types associated with variables `"z"` and `"s"`, respectively. The symbol `∋` (pronounced “ni”, for “in” backwards) is chosen because checking that `Γ ∋ x ⦂ A` is analogous to checking whether `x ⦂ A` appears in a list corresponding to `Γ`.

If two variables in a context have the same name, then lookup should return the most recently bound variable, which *shadows* the other variables. For example,

- `∅ , "x" ⦂ `ℕ ⇒ `ℕ , "x" ⦂ `ℕ ∋ "x" ⦂ `ℕ`.

Here `"x" ⦂ `ℕ ⇒ `ℕ` is shadowed by `"x" ⦂ `ℕ`.

Lookup is formalised as follows:

```
infix 4 _∋_⦂_

data _∋_⦂_ : Context → Id → Type → Set where

  Z : ∀ {Γ x A}
      ------------------
    → Γ , x ⦂ A ∋ x ⦂ A

  S : ∀ {Γ x y A B}
    → x ≢ y
    → Γ ∋ x ⦂ A
      ------------------
    → Γ , y ⦂ B ∋ x ⦂ A
```

The constructors `Z` and `S` correspond roughly to the constructors `here` and `there` for the element-of relation `_∈_` on lists. Constructor `S` takes an additional parameter, which ensures that when we look up a variable that it is not *shadowed* by another variable with the same name to its left in the list.

It can be rather tedious to use the `S` constructor, as you have to provide proofs that `x ≢ y` each time. For example:

```
_ : ∅ , "x" ⦂ `ℕ ⇒ `ℕ , "y" ⦂ `ℕ , "z" ⦂ `ℕ ∋ "x" ⦂ `ℕ ⇒ `ℕ
_ = S (λ()) (S (λ()) Z)
```

Instead, we’ll use a “smart constructor”, which uses proof by reflection to check the inequality while type checking:

```
S′ : ∀ {Γ x y A B}
   → {x≢y : False (x ≟ y)}
```

```
    → Γ ∋ x ⦂ A
      ------------------
    → Γ , y ⦂ B ∋ x ⦂ A

S′ {x≢y = x≢y} x = S (toWitnessFalse x≢y) x
```

Typing judgment

The second judgment is written

```
Γ ⊢ M ⦂ A
```

and indicates in context `Γ` that term `M` has type `A`. Context `Γ` provides types for all the free variables in `M`. For example:

- `∅ , "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`

Typing is formalised as follows:

```
infix 4 _⊢_⦂_

data _⊢_⦂_ : Context → Term → Type → Set where

  -- Axiom
  ⊢` : ∀ {Γ x A}
    → Γ ∋ x ⦂ A
      -----------
    → Γ ⊢ ` x ⦂ A

  -- ⇒-I
  ⊢ƛ : ∀ {Γ x N A B}
    → Γ , x ⦂ A ⊢ N ⦂ B
      -------------------
    → Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B

  -- ⇒-E
  _·_ : ∀ {Γ L M A B}
    → Γ ⊢ L ⦂ A ⇒ B
    → Γ ⊢ M ⦂ A
      -------------
    → Γ ⊢ L · M ⦂ B

  -- ℕ-I₁
  ⊢zero : ∀ {Γ}
      --------------
    → Γ ⊢ `zero ⦂ `ℕ

  -- ℕ-I₂
  ⊢suc : ∀ {Γ M}
    → Γ ⊢ M ⦂ `ℕ
      --------------
    → Γ ⊢ `suc M ⦂ `ℕ

  -- ℕ-E
  ⊢case : ∀ {Γ L M x N A}
    → Γ ⊢ L ⦂ `ℕ
    → Γ ⊢ M ⦂ A
    → Γ , x ⦂ `ℕ ⊢ N ⦂ A
      -------------------------------------
    → Γ ⊢ case L [zero⇒ M |suc x ⇒ N ] ⦂ A

  ⊢μ : ∀ {Γ x M A}
    → Γ , x ⦂ A ⊢ M ⦂ A
      -----------------
    → Γ ⊢ μ x ⇒ M ⦂ A
```

Each type rule is named after the constructor for the corresponding term.

Most of the rules have a second name, derived from a convention in logic, whereby the rule is named after the type connective that it concerns; rules to introduce and to eliminate each connective are labeled `-I` and `-E`, respectively. As we read the rules from top to bottom, introduction and elimination rules do what they say on the tin: the first *introduces* a formula for the connective, which appears in the conclusion but not in the premises; while the second *eliminates* a formula for the connective, which appears in a premise but not in the conclusion. An introduction rule describes how to construct a value of the type (abstractions yield functions, successor and zero yield naturals), while an elimination rule describes how to deconstruct a value of the given type (applications use functions, case expressions use naturals).

Note also the three places (in `⊢ƛ`, `⊢case`, and `⊢μ`) where the context is extended with `x` and an appropriate type, corresponding to the three places where a bound variable is introduced.

The rules are deterministic, in that at most one rule applies to every term.

Example type derivations

Type derivations correspond to trees. In informal notation, here is a type derivation for the Church numeral two,

```
                        ∋s                     ∋z
                        ------------------ ⊢`  -------------- ⊢`
∋s                      Γ₂ ⊢ ` "s" ⦂ A ⇒ A     Γ₂ ⊢ ` "z" ⦂ A
------------------ ⊢`   ------------------------------------- _·_
Γ₂ ⊢ ` "s" ⦂ A ⇒ A      Γ₂ ⊢ ` "s" · ` "z" ⦂ A
-------------------------------------------------- _·_
Γ₂ ⊢ ` "s" · (` "s" · ` "z") ⦂ A
-------------------------------------------- ⊢ƛ
Γ₁ ⊢ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") ⦂ A ⇒ A
------------------------------------------------------------- ⊢ƛ
Γ ⊢ ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z") ⦂ (A ⇒ A) ⇒ A ⇒ A
```

where `∋s` and `∋z` abbreviate the two derivations,

```
             ---------------- Z
"s" ≢ "z"    Γ₁ ∋ "s" ⦂ A ⇒ A
----------------------------- S       ------------- Z
Γ₂ ∋ "s" ⦂ A ⇒ A                       Γ₂ ∋ "z" ⦂ A
```

and where `Γ₁ = Γ , "s" ⦂ A ⇒ A` and `Γ₂ = Γ , "s" ⦂ A ⇒ A , "z" ⦂ A`. The typing derivation is valid for any `Γ` and `A`, for instance, we might take `Γ` to be `∅` and `A` to be `` `ℕ ``.

Here is the above typing derivation formalised in Agda:

```
Ch : Type → Type
Ch A = (A ⇒ A) ⇒ A ⇒ A

⊢twoᶜ : ∀ {Γ A} → Γ ⊢ twoᶜ ⦂ Ch A
⊢twoᶜ = ⊢ƛ (⊢ƛ (⊢` ∋s · (⊢` ∋s · ⊢` ∋z)))
  where
  ∋s = S′ Z
  ∋z = Z
```

Here are the typings corresponding to computing two plus two:

```
⊢two : ∀ {Γ} → Γ ⊢ two ⦂ `ℕ
⊢two = ⊢suc (⊢suc ⊢zero)

⊢plus : ∀ {Γ} → Γ ⊢ plus ⦂ `ℕ ⇒ `ℕ ⇒ `ℕ
⊢plus = ⊢μ (⊢ƛ (⊢ƛ (⊢case (⊢` ∋m) (⊢` ∋n)
         (⊢suc (⊢` ∋+ · ⊢` ∋m′ · ⊢` ∋n′)))))
  where
  ∋+ = S′ (S′ (S′ Z))
  ∋m = S′ Z
  ∋n = Z
  ∋m′ = Z
  ∋n′ = S′ Z

⊢2+2 : ∅ ⊢ plus · two · two ⦂ `ℕ
⊢2+2 = ⊢plus · ⊢two · ⊢two
```

In contrast to our earlier examples, here we have typed `two` and `plus` in an arbitrary context rather than the empty context; this makes it easy to use them inside other binding contexts as well as at the top level. Here the two lookup judgments `∋m` and `∋m′` refer to two different bindings of variables named `"m"`. In contrast, the two judgments `∋n` and `∋n′` both refer to the same binding of `"n"` but accessed in different contexts, the first where `"n"` is the last binding in the context, and the second after `"m"` is bound in the successor branch of the case.

And here are typings for the remainder of the Church example:

```
⊢plusᶜ : ∀ {Γ A} → Γ ⊢ plusᶜ ⦂ Ch A ⇒ Ch A ⇒ Ch A
⊢plusᶜ = ⊢ƛ (⊢ƛ (⊢ƛ (⊢ƛ (⊢` ∋m · ⊢` ∋s · (⊢` ∋n · ⊢` ∋s · ⊢` ∋z)))))
  where
  ∋m = S′ (S′ (S′ Z))
  ∋n = S′ (S′ Z)
  ∋s = S′ Z
  ∋z = Z

⊢sucᶜ : ∀ {Γ} → Γ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` ∋n))
  where
  ∋n = Z

⊢2+2ᶜ : ∅ ⊢ plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero ⦂ `ℕ
⊢2+2ᶜ = ⊢plusᶜ · ⊢twoᶜ · ⊢twoᶜ · ⊢sucᶜ · ⊢zero
```

Interaction with Agda

Construction of a type derivation may be done interactively. Start with the declaration:

```
⊢sucᶜ : ∅ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
⊢sucᶜ = ?
```

Typing C-c C-l causes Agda to create a hole and tell us its expected type:

```
⊢sucᶜ = { }0
?0 : ∅ ⊢ sucᶜ ⦂ `ℕ ⇒ `ℕ
```

Now we fill in the hole by typing C-c C-r. Agda observes that the outermost term in `sucᶜ` is `ƛ`, which is typed using `⊢ƛ`. The `⊢ƛ` rule in turn takes one argument, which Agda leaves as a hole:

```
⊢sucᶜ = ⊢ƛ { }1
?1 : ∅ , "n" ⦂ `ℕ ⊢ `suc ` "n" ⦂ `ℕ
```

We can fill in the hole by typing C-c C-r again:

```
⊢sucᶜ = ⊢ƛ (⊢suc { }2)
?2 : ∅ , "n" ⦂ `ℕ ⊢ ` "n" ⦂ `ℕ
```

And again:

```
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` { }3))
?3 : ∅ , "n" ⦂ `ℕ ∋ "n" ⦂ `ℕ
```

A further attempt with C-c C-r yields the message:

```
Don't know which constructor to introduce of Z or S
```

We can fill in `Z` by hand. If we type C-c C-space, Agda will confirm we are done:

```
⊢sucᶜ = ⊢ƛ (⊢suc (⊢` Z))
```

The entire process can be automated using Agsy, invoked with C-c C-a.

Chapter Inference will show how to use Agda to compute type derivations directly.

Lookup is injective

The lookup relation `Γ ∋ x ⦂ A` is injective, in that for each `Γ` and `x` there is at most one `A` such that the judgment holds:

```
∋-injective : ∀ {Γ x A B} → Γ ∋ x ⦂ A → Γ ∋ x ⦂ B → A ≡ B
∋-injective Z Z                = refl
∋-injective Z (S x≢ _)         = ⊥-elim (x≢ refl)
∋-injective (S x≢ _) Z         = ⊥-elim (x≢ refl)
∋-injective (S _ ∋x) (S _ ∋x′) = ∋-injective ∋x ∋x′
```

The typing relation `Γ ⊢ M ⦂ A` is not injective. For example, in any `Γ` the term `ƛ "x" ⇒ ` "x"` has type `A ⇒ A` for any type `A`.

Non-examples

We can also show that terms are *not* typeable. For example, here is a formal proof that it is not possible to type the term `` `zero · `suc `zero ``. It cannot be typed, because doing so requires that the first term in the application is both a natural and a function:

```
nope₁ : ∀ {A} → ¬ (∅ ⊢ `zero · `suc `zero ⦂ A)
nope₁ (() · _)
```

As a second example, here is a formal proof that it is not possible to type `` ƛ "x" ⇒ ` "x" · ` "x" ``. It cannot be typed, because doing so requires types `A` and `B` such that `A ⇒ B ≡ A`:

```
nope₂ : ∀ {A} → ¬ (∅ ⊢ ƛ "x" ⇒ ` "x" · ` "x" ⦂ A)
nope₂ (⊢ƛ (⊢` ∋x · ⊢` ∋x′)) = contradiction (∋-injective ∋x ∋x′)
  where
  contradiction : ∀ {A B} → ¬ (A ⇒ B ≡ A)
  contradiction ()
```

Quiz

For each of the following, give a type `A` for which it is derivable, or explain why there is no such `A`.

1. `` ∅ , "y" ⦂ `ℕ ⇒ `ℕ , "x" ⦂ `ℕ ⊢ ` "y" · ` "x" ⦂ A ``
2. `` ∅ , "y" ⦂ `ℕ ⇒ `ℕ , "x" ⦂ `ℕ ⊢ ` "x" · ` "y" ⦂ A ``
3. `` ∅ , "y" ⦂ `ℕ ⇒ `ℕ ⊢ ƛ "x" ⇒ ` "y" · ` "x" ⦂ A ``

For each of the following, give types `A`, `B`, and `C` for which it is derivable, or explain why there are no such types.

1. `` ∅ , "x" ⦂ A ⊢ ` "x" · ` "x" ⦂ B ``
2. `` ∅ , "x" ⦂ A , "y" ⦂ B ⊢ ƛ "z" ⇒ ` "x" · (` "y" · ` "z") ⦂ C ``

Exercise `⊢mul` (recommended)

Using the term `mul` you defined earlier, write out the derivation showing that it is well typed.

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

Exercise `⊢mulᶜ` (practice)

Using the term `mulᶜ` you defined earlier, write out the derivation showing that it is well typed.

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

Unicode

This chapter uses the following unicode:

```
⇒  U+21D2  RIGHTWARDS DOUBLE ARROW (\=>)
ƛ  U+019B  LATIN SMALL LETTER LAMBDA WITH STROKE (\Gl-)
·  U+00B7  MIDDLE DOT (\cdot)
≟  U+225F  QUESTIONED EQUAL TO (\?=)
—  U+2014  EM DASH (\em)
↠  U+21A0  RIGHTWARDS TWO HEADED ARROW (\rr-)
ξ  U+03BE  GREEK SMALL LETTER XI (\Gx or \xi)
β  U+03B2  GREEK SMALL LETTER BETA (\Gb or \beta)
Γ  U+0393  GREEK CAPITAL LETTER GAMMA (\GG or \Gamma)
≠  U+2260  NOT EQUAL TO (\=n or \ne)
∋  U+220B  CONTAINS AS MEMBER (\ni)
∅  U+2205  EMPTY SET (\0)
⊢  U+22A2  RIGHT TACK (\vdash or \|-)
⦂  U+2982  Z NOTATION TYPE COLON (\:)
😇  U+1F607  SMILING FACE WITH HALO
😈  U+1F608  SMILING FACE WITH HORNS
```

We compose reduction `—→` from an em dash `—` and an arrow `→`. Similarly for reflexive and transitive closure `—↠`.

Chapter 12

Properties: Progress and Preservation

```
module plfa.part2.Properties where
```

This chapter covers properties of the simply-typed lambda calculus, as introduced in the previous chapter. The most important of these properties are progress and preservation. We introduce these below, and show how to combine them to get Agda to compute reduction sequences for us.

Imports

```
open import Relation.Binary.PropositionalEquality
  using (_≡_; _≢_; refl; sym; cong; cong₂)
open import Data.String using (String; _≟_)
open import Data.Nat using (ℕ; zero; suc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product
  using (_×_; proj₁; proj₂; ∃; ∃-syntax)
  renaming (_,_ to ⟨_,_⟩)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Relation.Nullary using (¬_; Dec; yes; no)
open import Function using (_∘_)
open import plfa.part1.Isomorphism
open import plfa.part2.Lambda
```

Introduction

The last chapter introduced simply-typed lambda calculus, including the notions of closed terms, terms that are values, reducing one term to another, and well-typed terms.

Ultimately, we would like to show that we can keep reducing a term until we reach a value. For instance, in the last chapter we showed that two plus two is four,

```
plus · two · two  —↠  `suc `suc `suc `suc `zero
```

which was proved by a long chain of reductions, ending in the value on the right. Every term in the chain had the same type, `` `ℕ ``. We also saw a second, similar example involving Church numerals.

What we might expect is that every term is either a value or can take a reduction step. As we will see, this property does *not* hold for every term, but it does hold for every closed, well-typed term.

Progress: If `∅ ⊢ M ⦂ A` then either `M` is a value or there is an `N` such that `M —→ N`.

So, either we have a value, and we are done, or we can take a reduction step. In the latter case, we would like to apply progress again. But to do so we need to know that the term yielded by the reduction is itself closed and well typed. It turns out that this property holds whenever we start with a closed, well-typed term.

Preservation: If `∅ ⊢ M ⦂ A` and `M —→ N` then `∅ ⊢ N ⦂ A`.

This gives us a recipe for automating evaluation. Start with a closed and well-typed term. By progress, it is either a value, in which case we are done, or it reduces to some other term. By preservation, that other term will itself be closed and well typed. Repeat. We will either loop forever, in which case evaluation does not terminate, or we will eventually reach a value, which is guaranteed to be closed and of the same type as the original term. We will turn this recipe into Agda code that can compute for us the reduction sequence of `plus · two · two`, and its Church numeral variant.

(The development in this chapter was inspired by the corresponding development in *Software Foundations*, Volume *Programming Language Foundations*, Chapter *StlcProp*. It will turn out that one of our technical choices — to introduce an explicit judgment `Γ ∋ x ⦂ A` in place of treating a context as a function from identifiers to types — permits a simpler development. In particular, we can prove substitution preserves types without needing to develop a separate inductive definition of the `appears_free_in` relation.)

Values do not reduce

We start with an easy observation. Values do not reduce:

```
V¬—→ : ∀ {M N}
  → Value M
    ----------
  → ¬ (M —→ N)
V¬—→ V-ƛ    ()
V¬—→ V-zero ()
V¬—→ (V-suc VM) (ξ-suc M—→N) = V¬—→ VM M—→N
```

We consider the three possibilities for values:

- If it is an abstraction then no reduction applies
- If it is zero then no reduction applies
- If it is a successor then rule `ξ-suc` may apply, but in that case the successor is itself of a value that reduces, which by induction cannot occur.

As a corollary, terms that reduce are not values:

```
—→¬V : ∀ {M N}
  → M —→ N
    ---------
  → ¬ Value M
—→¬V M—→N VM = V¬—→ VM M—→N
```

If we expand out the negations, we have

```
V¬—→ : ∀ {M N} → Value M → M —→ N → ⊥
—→¬V : ∀ {M N} → M —→ N → Value M → ⊥
```

which are the same function with the arguments swapped.

Canonical Forms

Well-typed values must take one of a small number of *canonical forms*, which provide an analogue of the `Value` relation that relates values to their types. A lambda expression must have a function type, and a zero or successor expression must be a natural. Further, the body of a function must be well typed in a context containing only its bound variable, and the argument of successor must itself be canonical:

```
infix 4 Canonical_⦂_

data Canonical_⦂_ : Term → Type → Set where

  C-ƛ : ∀ {x A N B}
    → ∅ , x ⦂ A ⊢ N ⦂ B
      -----------------------------
    → Canonical (ƛ x ⇒ N) ⦂ (A ⇒ B)

  C-zero :
      --------------------
      Canonical `zero ⦂ `ℕ

  C-suc : ∀ {V}
    → Canonical V ⦂ `ℕ
      ---------------------
    → Canonical `suc V ⦂ `ℕ
```

Every closed, well-typed value is canonical:

```
canonical : ∀ {V A}
  → ∅ ⊢ V ⦂ A
  → Value V
    -----------
  → Canonical V ⦂ A
canonical (⊢` ())   ()
canonical (⊢ƛ ⊢N)   V-ƛ        = C-ƛ ⊢N
canonical (⊢L · ⊢M) ()
canonical ⊢zero     V-zero     = C-zero
canonical (⊢suc ⊢V) (V-suc VV) = C-suc (canonical ⊢V VV)
canonical (⊢case ⊢L ⊢M ⊢N) ()
canonical (⊢μ ⊢M)   ()
```

There are only three interesting cases to consider:

- If the term is a lambda abstraction, then well-typing of the term guarantees well-typing of the body.
- If the term is zero then it is canonical trivially.
- If the term is a successor then since it is well typed its argument is well typed, and since it is a value its argument is a value. Hence, by induction its argument is also canonical.

The variable case is thrown out because a closed term has no free variables and because a variable is not a value. The cases for application, case expression, and fixpoint are thrown out because they are not values.

Conversely, if a term is canonical then it is a value and it is well typed in the empty context:

```
value : ∀ {M A}
  → Canonical M ⦂ A
    ----------------
  → Value M
value (C-ƛ ⊢N)    = V-ƛ
value C-zero      = V-zero
value (C-suc CM) = V-suc (value CM)

typed : ∀ {M A}
  → Canonical M ⦂ A
    ---------------
  → ∅ ⊢ M ⦂ A
typed (C-ƛ ⊢N)    = ⊢ƛ ⊢N
typed C-zero      = ⊢zero
typed (C-suc CM) = ⊢suc (typed CM)
```

The proofs are straightforward, and again use induction in the case of successor.

Progress

We would like to show that every term is either a value or takes a reduction step. However, this is not true in general. The term

```
`zero · `suc `zero
```

is neither a value nor can take a reduction step. And if `s ⦂ `ℕ ⇒ `ℕ` then the term

```
s · `zero
```

cannot reduce because we do not know which function is bound to the free variable `s`. The first of those terms is ill typed, and the second has a free variable. Every term that is well typed and closed has the desired property.

Progress: If `∅ ⊢ M ⦂ A` then either `M` is a value or there is an `N` such that `M —→ N`.

To formulate this property, we first introduce a relation that captures what it means for a term `M` to make progress:

```
data Progress (M : Term) : Set where

  step : ∀ {N}
    → M —→ N
      ----------
    → Progress M

  done :
      Value M
      ----------
    → Progress M
```

A term `M` makes progress if either it can take a step, meaning there exists a term `N` such that `M —→ N`, or if it is done, meaning that `M` is a value.

If a term is well typed in the empty context then it satisfies progress:

```
progress : ∀ {M A}
  → ∅ ⊢ M ⦂ A
    ----------
  → Progress M
progress (⊢` ())
progress (⊢ƛ ⊢N) = done V-ƛ
progress (⊢L · ⊢M) with progress ⊢L
... | step L—→L′ = step (ξ-·₁ L—→L′)
... | done VL with progress ⊢M
... | step M—→M′ = step (ξ-·₂ VL M—→M′)
... | done VM with canonical ⊢L VL
... | C-ƛ _        = step (β-ƛ VM)
progress ⊢zero   = done V-zero
progress (⊢suc ⊢M) with progress ⊢M
... | step M—→M′ = step (ξ-suc M—→M′)
... | done VM      = done (V-suc VM)
progress (⊢case ⊢L ⊢M ⊢N) with progress ⊢L
... | step L—→L′ = step (ξ-case L—→L′)
... | done VL with canonical ⊢L VL
... | C-zero      = step β-zero
... | C-suc CL    = step (β-suc (value CL))
progress (⊢μ ⊢M) = step β-μ
```

We induct on the evidence that the term is well typed. Let’s unpack the first three cases:

- The term cannot be a variable, since no variable is well typed in the empty context.
- If the term is a lambda abstraction then it is a value.
- If the term is an application `L · M`, recursively apply progress to the derivation that `L` is well typed:
  - If the term steps, we have evidence that `L —→ L′`, which by `ξ-·₁` means that our original term steps to `L′ · M`
  - If the term is done, we have evidence that `L` is a value. Recursively apply progress to the derivation that `M` is well typed:
    * If the term steps, we have evidence that `M —→ M′`, which by `ξ-·₂` means that our original term steps to `L · M′`. Step `ξ-·₂` applies only if we have evidence that `L` is a value, but progress on that subterm has already supplied the required evidence.
    * If the term is done, we have evidence that `M` is a value. We apply the canonical forms lemma to the evidence that `L` is well typed and a value, which since we are in an application leads to the conclusion that `L` must be a lambda abstraction. We also have evidence that `M` is a value, so our original term steps by `β-ƛ`.

The remaining cases are similar. If by induction we have a `step` case we apply a `ξ` rule, and if we have a `done` case then either we have a value or apply a `β` rule. For fixpoint, no induction is required as the `β` rule applies immediately.

Our code reads neatly in part because we consider the `step` option before the `done` option. We could, of course, do it the other way around, but then the `...` abbreviation no longer works, and we will need to write out all the arguments in full. In general, the rule of thumb is to consider the easy case (here `step`) before the hard case (here `done`). If you have two hard cases, you will have to expand out `...` or introduce subsidiary functions.

Instead of defining a data type for `Progress M`, we could have formulated progress using disjunction and existentials:

```
postulate
  progress′ : ∀ M {A} → ∅ ⊢ M ⦂ A → Value M ⊎ ∃[ N ](M —→ N)
```

This leads to a less perspicuous proof. Instead of the mnemonic `done` and `step` we use `inj₁` and `inj₂`, and the term `N` is no longer implicit and so must be written out in full. In the case for `β-ƛ` this requires that we match against the lambda expression `L` to determine its bound variable and body, `ƛ x ⇒ N`, so we can show that `L · M` reduces to `N [ x := M ]`.

Exercise `Progress-≃` **(practice)**

Show that `Progress M` is isomorphic to `Value M ⊎ ∃[ N ](M —→ N)`.

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

Exercise `progress′` **(practice)**

Write out the proof of `progress′` in full, and compare it to the proof of `progress` above.

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

Exercise `value?` **(practice)**

Combine `progress` and `—→¬V` to write a program that decides whether a well-typed term is a value:

```
postulate
  value? : ∀ {A M} → ∅ ⊢ M ⦂ A → Dec (Value M)
```

Prelude to preservation

The other property we wish to prove, preservation of typing under reduction, turns out to require considerably more work. The proof has three key steps.

The first step is to show that types are preserved by *renaming*.

Renaming: Let `Γ` and `Δ` be two contexts such that every variable that appears in `Γ` also appears with the same type in `Δ`. Then if any term is typeable under `Γ`, it has the same type under `Δ`.

In symbols:

```
∀ {x A} → Γ ∋ x ⦂ A  →  Δ ∋ x ⦂ A
---------------------------------
∀ {M A} → Γ ⊢ M ⦂ A  →  Δ ⊢ M ⦂ A
```

Three important corollaries follow. The *weaken* lemma asserts that a term which is well typed in the empty context is also well typed in an arbitrary context. The *drop* lemma asserts that a term which is well typed in a context where the same variable appears twice remains well typed if we drop the shadowed occurrence. The *swap* lemma asserts that a term which is well typed in a context remains well typed if we swap two variables.

(Renaming is similar to the *context invariance* lemma in *Software Foundations*, but it does not require the definition of `appears_free_in` nor the `free_in_context` lemma.)

The second step is to show that types are preserved by *substitution*.

Substitution: Say we have a closed term `V` of type `A`, and under the assumption that `x` has type `A` the term `N` has type `B`. Then substituting `V` for `x` in `N` yields a term that also has type `B`.

In symbols:

```
∅ ⊢ V ⦂ A
Γ , x ⦂ A ⊢ N ⦂ B
--------------------
Γ ⊢ N [ x := V ] ⦂ B
```

The result does not depend on `V` being a value, but it does require that `V` be closed; recall that we restricted our attention to substitution by closed terms in order to avoid the need to rename bound variables. The term into which we are substituting is typed in an arbitrary context `Γ`, extended by the variable `x` for which we are substituting; and the result term is typed in `Γ`.

The lemma establishes that substitution composes well with typing: typing the components separately guarantees that the result of combining them is also well typed.

The third step is to show preservation.

Preservation: If `∅ ⊢ M ⦂ A` and `M —→ N` then `∅ ⊢ N ⦂ A`.

The proof is by induction over the possible reductions, and the substitution lemma is crucial in showing that each of the `β` rules that uses substitution preserves types.

We now proceed with our three-step programme.

Renaming

We often need to “rebase” a type derivation, replacing a derivation `Γ ⊢ M ⦂ A` by a related derivation `Δ ⊢ M ⦂ A`. We may do so as long as every variable that appears in `Γ` also appears in `Δ`, and with the same type.

Three of the rules for typing (lambda abstraction, case on naturals, and fixpoint) have hypotheses that extend the context to include a bound variable. In each of these rules, `Γ` appears in the conclusion and `Γ , x ⦂ A` appears in a hypothesis. Thus:

```
Γ , x ⦂ A ⊢ N ⦂ B
------------------- ⊢ƛ
Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B
```

for lambda expressions, and similarly for case and fixpoint. To deal with this situation, we first prove a lemma showing that if one context maps to another, this is still true after adding the same variable to both contexts:

```
ext : ∀ {Γ Δ}
  → (∀ {x A} → Γ ∋ x ⦂ A → Δ ∋ x ⦂ A)
    -----------------------------------------------------
  → (∀ {x y A B} → Γ , y ⦂ B ∋ x ⦂ A → Δ , y ⦂ B ∋ x ⦂ A)
ext ρ Z           = Z
ext ρ (S x≢y ∋x) = S x≢y (ρ ∋x)
```

Let `ρ` be the name of the map that takes evidence that `x` appears in `Γ` to evidence that `x` appears in `Δ`. The proof is by case analysis of the evidence that `x` appears in the extended map `Γ , y ⦂ B`:

- If `x` is the same as `y`, we used `Z` to access the last variable in the extended `Γ`; and can similarly use `Z` to access the last variable in the extended `Δ`.
- If `x` differs from `y`, then we used `S` to skip over the last variable in the extended `Γ`, where `x≢y` is evidence that `x` and `y` differ, and `∋x` is the evidence that `x` appears in `Γ`; and we can similarly use `S` to skip over the last variable in the extended `Δ`, applying `ρ` to find the evidence that `x` appears in `Δ`.

With the extension lemma under our belts, it is straightforward to prove renaming preserves types:

```
rename : ∀ {Γ Δ}
  → (∀ {x A} → Γ ∋ x ⦂ A → Δ ∋ x ⦂ A)
    ----------------------------------
  → (∀ {M A} → Γ ⊢ M ⦂ A → Δ ⊢ M ⦂ A)
rename ρ (⊢` ∋w)           = ⊢` (ρ ∋w)
rename ρ (⊢ƛ ⊢N)           = ⊢ƛ (rename (ext ρ) ⊢N)
rename ρ (⊢L · ⊢M)         = (rename ρ ⊢L) · (rename ρ ⊢M)
rename ρ ⊢zero             = ⊢zero
rename ρ (⊢suc ⊢M)         = ⊢suc (rename ρ ⊢M)
rename ρ (⊢case ⊢L ⊢M ⊢N) = ⊢case (rename ρ ⊢L) (rename ρ ⊢M) (rename (ext ρ) ⊢N)
rename ρ (⊢μ ⊢M)           = ⊢μ (rename (ext ρ) ⊢M)
```

As before, let `ρ` be the name of the map that takes evidence that `x` appears in `Γ` to evidence that `x` appears in `Δ`. We induct on the evidence that `M` is well typed in `Γ`. Let’s unpack the first three cases:

- If the term is a variable, then applying `ρ` to the evidence that the variable appears in `Γ` yields the corresponding evidence that the variable appears in `Δ`.
- If the term is a lambda abstraction, use the previous lemma to extend the map `ρ` suitably and use induction to rename the body of the abstraction.
- If the term is an application, use induction to rename both the function and the argument.

The remaining cases are similar, using induction for each subterm, and extending the map whenever the construct introduces a bound variable.

The induction is over the derivation that the term is well typed, so extending the context doesn't invalidate the inductive hypothesis. Equivalently, the recursion terminates because the second argument always grows smaller, even though the first argument sometimes grows larger.

We have three important corollaries, each proved by constructing a suitable map between contexts.

First, a closed term can be weakened to any context:

```
weaken : ∀ {Γ M A}
  → ∅ ⊢ M ⦂ A
    ----------
  → Γ ⊢ M ⦂ A
weaken {Γ} ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → ∅ ∋ z ⦂ C
      ---------
    → Γ ∋ z ⦂ C
  ρ ()
```

Here the map `ρ` is trivial, since there are no possible arguments in the empty context `∅`.

Second, if the last two variables in a context are equal then we can drop the shadowed one:

```
drop : ∀ {Γ x M A B C}
  → Γ , x ⦂ A , x ⦂ B ⊢ M ⦂ C
    --------------------------
  → Γ , x ⦂ B ⊢ M ⦂ C
drop {Γ} {x} {M} {A} {B} {C} ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → Γ , x ⦂ A , x ⦂ B ∋ z ⦂ C
      -------------------------
    → Γ , x ⦂ B ∋ z ⦂ C
  ρ Z                 = Z
  ρ (S x≢x Z)         = ⊥-elim (x≢x refl)
  ρ (S z≢x (S _ ∋z)) = S z≢x ∋z
```

Here map `ρ` can never be invoked on the inner occurrence of `x` since it is masked by the outer occurrence. Skipping over the `x` in the first position can only happen if the variable looked for differs from `x` (the evidence for which is `x≢x` or `z≢x`) but if the variable is found in the second position, which also contains `x`, this leads to a contradiction (evidenced by `x≢x refl`).

Third, if the last two variables in a context differ then we can swap them:

```
swap : ∀ {Γ x y M A B C}
  → x ≢ y
  → Γ , y ⦂ B , x ⦂ A ⊢ M ⦂ C
    --------------------------
  → Γ , x ⦂ A , y ⦂ B ⊢ M ⦂ C
swap {Γ} {x} {y} {M} {A} {B} {C} x≢y ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → Γ , y ⦂ B , x ⦂ A ∋ z ⦂ C
      --------------------------
    → Γ , x ⦂ A , y ⦂ B ∋ z ⦂ C
  ρ Z                  = S x≢y Z
  ρ (S z≢x Z)          = Z
  ρ (S z≢x (S z≢y ∋z)) = S z≢y (S z≢x ∋z)
```

Here the renaming map takes a variable at the end into a variable one from the end, and vice versa. The first line is responsible for moving `x` from a position at the end to a position one from the end with `y` at the end, and requires the provided evidence that `x ≢ y`.

Substitution

The key to preservation – and the trickiest bit of the proof – is the lemma establishing that substitution preserves types.

Recall that in order to avoid renaming bound variables, substitution is restricted to be by closed terms only. This restriction was not enforced by our definition of substitution, but it is captured by our lemma to assert that substitution preserves typing.

Our concern is with reducing closed terms, which means that when we apply `β` reduction, the term substituted in contains a single free variable (the bound variable of the lambda abstraction, or similarly for case or fixpoint). However, substitution is defined by recursion, and as we descend into terms with bound variables the context grows. So for the induction to go through, we require an arbitrary context `Γ`, as in the statement of the lemma.

Here is the formal statement and proof that substitution preserves types:

```
subst : ∀ {Γ x N V A B}
  → ∅ ⊢ V ⦂ A
  → Γ , x ⦂ A ⊢ N ⦂ B
    --------------------
  → Γ ⊢ N [ x := V ] ⦂ B
subst {x = y} ⊢V (⊢` {x = x} Z) with x ≟ y
... | yes _          = weaken ⊢V
... | no  x≢y        = ⊥-elim (x≢y refl)
subst {x = y} ⊢V (⊢` {x = x} (S x≢y ∋x)) with x ≟ y
... | yes refl       = ⊥-elim (x≢y refl)
... | no  _          = ⊢` ∋x
subst {x = y} ⊢V (⊢ƛ {x = x} ⊢N) with x ≟ y
... | yes refl       = ⊢ƛ (drop ⊢N)
... | no  x≢y        = ⊢ƛ (subst ⊢V (swap x≢y ⊢N))
subst ⊢V (⊢L · ⊢M) = (subst ⊢V ⊢L) · (subst ⊢V ⊢M)
subst ⊢V ⊢zero      = ⊢zero
subst ⊢V (⊢suc ⊢M) = ⊢suc (subst ⊢V ⊢M)
subst {x = y} ⊢V (⊢case {x = x} ⊢L ⊢M ⊢N) with x ≟ y
... | yes refl       = ⊢case (subst ⊢V ⊢L) (subst ⊢V ⊢M) (drop ⊢N)
... | no  x≢y        = ⊢case (subst ⊢V ⊢L) (subst ⊢V ⊢M) (subst ⊢V (swap x≢y ⊢N))
subst {x = y} ⊢V (⊢μ {x = x} ⊢M) with x ≟ y
```

```
... | yes refl      = ⊢μ (drop ⊢M)
... | no x≢y        = ⊢μ (subst ⊢V (swap x≢y ⊢M))
```

We induct on the evidence that `N` is well typed in the context `Γ` extended by `x`.

First, we note a wee issue with naming. In the lemma statement, the variable `x` is an implicit parameter for the variable substituted, while in the type rules for variables, abstractions, cases, and fixpoints, the variable `x` is an implicit parameter for the relevant variable. We are going to need to get hold of both variables, so we use the syntax `{x = y}` to bind `y` to the substituted variable and the syntax `{x = x}` to bind `x` to the relevant variable in the patterns for `⊢``, `⊢ƛ`, `⊢case`, and `⊢μ`. Using the name `y` here is consistent with the naming in the original definition of substitution in the previous chapter. The proof never mentions the types of `x`, `y`, `V`, or `N`, so in what follows we choose type names as convenient.

Now that naming is resolved, let's unpack the first three cases:

- In the variable case, we must show

  ```
  ∅ ⊢ V ⦂ B
  Γ , y ⦂ B ⊢ ` x ⦂ A
  ------------------------
  Γ ⊢ ` x [ y := V ] ⦂ A
  ```

  where the second hypothesis follows from:

  ```
  Γ , y ⦂ B ∋ x ⦂ A
  ```

  There are two subcases, depending on the evidence for this judgment:

  - The lookup judgment is evidenced by rule `Z`:

    ```
    ----------------
    Γ , x ⦂ A ∋ x ⦂ A
    ```

    In this case, `x` and `y` are necessarily identical, as are `A` and `B`. Nonetheless, we must evaluate `x ≟ y` in order to allow the definition of substitution to simplify:

    ∗ If the variables are equal, then after simplification we must show

      ```
      ∅ ⊢ V ⦂ A
      ---------
      Γ ⊢ V ⦂ A
      ```

      which follows by weakening.

    ∗ If the variables are unequal we have a contradiction.

  - The lookup judgment is evidenced by rule `S`:

    ```
    x ≢ y
    Γ ∋ x ⦂ A
    -----------------
    Γ , y ⦂ B ∋ x ⦂ A
    ```

    In this case, `x` and `y` are necessarily distinct. Nonetheless, we must again evaluate `x ≟ y` in order to allow the definition of substitution to simplify:

    ∗ If the variables are equal we have a contradiction.

    ∗ If the variables are unequal, then after simplification we must show

```
∅ ⊢ V ⦂ B
x ≢ y
Γ ∋ x ⦂ A
-------------
Γ ⊢ ` x ⦂ A
```

which follows by the typing rule for variables.

- In the abstraction case, we must show

```
∅ ⊢ V ⦂ B
Γ , y ⦂ B ⊢ (ƛ x ⇒ N) ⦂ A ⇒ C
--------------------------------
Γ ⊢ (ƛ x ⇒ N) [ y := V ] ⦂ A ⇒ C
```

where the second hypothesis follows from

```
Γ , y ⦂ B , x ⦂ A ⊢ N ⦂ C
```

We evaluate `x ≟ y` in order to allow the definition of substitution to simplify:

  - If the variables are equal then after simplification we must show:

```
∅ ⊢ V ⦂ B
Γ , x ⦂ B , x ⦂ A ⊢ N ⦂ C
-------------------------
Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ C
```

  From the drop lemma, `drop`, we may conclude:

```
Γ , x ⦂ B , x ⦂ A ⊢ N ⦂ C
-------------------------
Γ , x ⦂ A ⊢ N ⦂ C
```

  The typing rule for abstractions then yields the required conclusion.

  - If the variables are distinct then after simplification we must show:

```
∅ ⊢ V ⦂ B
Γ , y ⦂ B , x ⦂ A ⊢ N ⦂ C
--------------------------------
Γ ⊢ ƛ x ⇒ (N [ y := V ]) ⦂ A ⇒ C
```

  From the swap lemma we may conclude:

```
Γ , y ⦂ B , x ⦂ A ⊢ N ⦂ C
-------------------------
Γ , x ⦂ A , y ⦂ B ⊢ N ⦂ C
```

  The inductive hypothesis gives us:

```
∅ ⊢ V ⦂ B
Γ , x ⦂ A , y ⦂ B ⊢ N ⦂ C
----------------------------
Γ , x ⦂ A ⊢ N [ y := V ] ⦂ C
```

  The typing rule for abstractions then yields the required conclusion.

- In the application case, we must show

```
∅ ⊢ V ⦂ C
Γ , y ⦂ C ⊢ L · M ⦂ B
-------------------------
Γ ⊢ (L · M) [ y := V ] ⦂ B
```

where the second hypothesis follows from the two judgments

```
Γ , y ⦂ C ⊢ L ⦂ A ⇒ B
Γ , y ⦂ C ⊢ M ⦂ A
```

By the definition of substitution, we must show:

```
∅ ⊢ V ⦂ C
Γ , y ⦂ C ⊢ L ⦂ A ⇒ B
Γ , y ⦂ C ⊢ M ⦂ A
---------------------------------------
Γ ⊢ (L [ y := V ]) · (M [ y := V ]) ⦂ B
```

Applying the induction hypothesis for `L` and `M` and the typing rule for applications yields the required conclusion.

The remaining cases are similar, using induction for each subterm. Where the construct introduces a bound variable we need to compare it with the substituted variable, applying the drop lemma if they are equal and the swap lemma if they are distinct.

For Agda it makes a difference whether we write `x ≟ y` or `y ≟ x`. In an interactive proof, Agda will show which residual `with` clauses in the definition of `_[_:=_]` need to be simplified, and the `with` clauses in `subst` need to match these exactly. The guideline is that Agda knows nothing about symmetry or commutativity, which require invoking appropriate lemmas, so it is important to think about order of arguments and to be consistent.

Exercise `subst′` (stretch)

Rewrite `subst` to work with the modified definition `_[_:=_]′` from the exercise in the previous chapter. As before, this should factor dealing with bound variables into a single function, defined by mutual recursion with the proof that substitution preserves types.

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

Preservation

Once we have shown that substitution preserves types, showing that reduction preserves types is straightforward:

```
preserve : ∀ {M N A}
  → ∅ ⊢ M ⦂ A
  → M —→ N
    ----------
  → ∅ ⊢ N ⦂ A
preserve (⊢` ())
preserve (⊢ƛ ⊢N)                 ()
preserve (⊢L · ⊢M)            (ξ-·₁ L⟶L′)    = (preserve ⊢L L⟶L′) · ⊢M
preserve (⊢L · ⊢M)            (ξ-·₂ VL M⟶M′) = ⊢L · (preserve ⊢M M⟶M′)
preserve ((⊢ƛ ⊢N) · ⊢V)       (β-ƛ VV)       = subst ⊢V ⊢N
preserve ⊢zero                ()
preserve (⊢suc ⊢M)            (ξ-suc M⟶M′)   = ⊢suc (preserve ⊢M M⟶M′)
preserve (⊢case ⊢L ⊢M ⊢N)     (ξ-case L⟶L′)  = ⊢case (preserve ⊢L L⟶L′) ⊢M ⊢N
preserve (⊢case ⊢zero ⊢M ⊢N)  (β-zero)       = ⊢M
preserve (⊢case (⊢suc ⊢V) ⊢M ⊢N) (β-suc VV)  = subst ⊢V ⊢N
preserve (⊢μ ⊢M)              (β-μ)          = subst (⊢μ ⊢M) ⊢M
```

The proof never mentions the types of `M` or `N`, so in what follows we choose type name as convenient.

Let's unpack the cases for two of the reduction rules:

- Rule `ξ-·₁`. We have

  ```
  L ⟶ L′
  ----------------
  L · M ⟶ L′ · M
  ```

  where the left-hand side is typed by

  ```
  Γ ⊢ L ⦂ A ⇒ B
  Γ ⊢ M ⦂ A
  -------------
  Γ ⊢ L · M ⦂ B
  ```

  By induction, we have

  ```
  Γ ⊢ L ⦂ A ⇒ B
  L ⟶ L′
  --------------
  Γ ⊢ L′ ⦂ A ⇒ B
  ```

  from which the typing of the right-hand side follows immediately.

- Rule `β-ƛ`. We have

  ```
  Value V
  ----------------------------
  (ƛ x ⇒ N) · V ⟶ N [ x := V ]
  ```

  where the left-hand side is typed by

  ```
  Γ , x ⦂ A ⊢ N ⦂ B
  -------------------
  Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B    Γ ⊢ V ⦂ A
  --------------------------------
  Γ ⊢ (ƛ x ⇒ N) · V ⦂ B
  ```

  By the substitution lemma, we have

  ```
  Γ ⊢ V ⦂ A
  Γ , x ⦂ A ⊢ N ⦂ B
  --------------------
  Γ ⊢ N [ x := V ] ⦂ B
  ```

  from which the typing of the right-hand side follows immediately.

The remaining cases are similar. Each `ξ` rule follows by induction, and each `β` rule follows by the substitution lemma.

Evaluation

By repeated application of progress and preservation, we can evaluate any well-typed term. In this section, we will present an Agda function that computes the reduction sequence from any given closed, well-typed term to its value, if it has one.

Some terms may reduce forever. Here is a simple example:

```
sucμ = μ "x" ⇒ `suc (` "x")

_ =
  begin
    sucμ
  —→⟨ β-μ ⟩
    `suc sucμ
  —→⟨ ξ-suc β-μ ⟩
    `suc `suc sucμ
  —→⟨ ξ-suc (ξ-suc β-μ) ⟩
    `suc `suc `suc sucμ
  -- ...
  ∎
```

Since every Agda computation must terminate, we cannot simply ask Agda to reduce a term to a value. Instead, we will provide a natural number to Agda, and permit it to stop short of a value if the term requires more than the given number of reduction steps.

A similar issue arises with cryptocurrencies. Systems which use smart contracts require the miners that maintain the blockchain to evaluate the program which embodies the contract. For instance, validating a transaction on Ethereum may require executing a program for the Ethereum Virtual Machine (EVM). A long-running or non-terminating program might cause the miner to invest arbitrary effort in validating a contract for little or no return. To avoid this situation, each transaction is accompanied by an amount of *gas* available for computation. Each step executed on the EVM is charged an advertised amount of gas, and the transaction pays for the gas at a published rate: a given number of Ethers (the currency of Ethereum) per unit of gas.

By analogy, we will use the name *gas* for the parameter which puts a bound on the number of reduction steps. `Gas` is specified by a natural number:

```
record Gas : Set where
  constructor gas
  field
    amount : ℕ
```

When our evaluator returns a term `N`, it will either give evidence that `N` is a value or indicate that it ran out of gas:

```
data Finished (N : Term) : Set where

  done :
    Value N
    ----------
    → Finished N

  out-of-gas :
```

```
  ----------
  Finished N
```

Given a term `L` of type `A`, the evaluator will, for some `N`, return a reduction sequence from `L` to `N` and an indication of whether reduction finished:

```
data Steps (L : Term) : Set where

  steps : ∀ {N}
    → L —↠ N
    → Finished N
      ----------
    → Steps L
```

The evaluator takes gas and evidence that a term is well typed, and returns the corresponding steps:

```
eval : ∀ {L A}
  → Gas
  → ∅ ⊢ L ⦂ A
    ---------
  → Steps L
eval {L} (gas zero) ⊢L = steps (L ∎) out-of-gas
eval {L} (gas (suc m)) ⊢L with progress ⊢L
... | done VL          = steps (L ∎) (done VL)
... | step {M} L—→M with eval (gas m) (preserve ⊢L L—→M)
... | steps M—↠N fin   = steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

Let `L` be the name of the term we are reducing, and `⊢L` be the evidence that `L` is well typed. We consider the amount of gas remaining. There are two possibilities:

- It is zero, so we stop early. We return the trivial reduction sequence `L —↠ L`, evidence that `L` is well typed, and an indication that we are out of gas.
- It is non-zero and after the next step we have `m` gas remaining. Apply progress to the evidence that term `L` is well typed. There are two possibilities:
  - Term `L` is a value, so we are done. We return the trivial reduction sequence `L —↠ L`, evidence that `L` is well typed, and the evidence that `L` is a value.
  - Term `L` steps to another term `M`. Preservation provides evidence that `M` is also well typed, and we recursively invoke `eval` on the remaining gas. The result is evidence that `M —↠ N`, together with evidence that `N` is well typed and an indication of whether reduction finished. We combine the evidence that `L —→ M` and `M —↠ N` to return evidence that `L —↠ N`, together with the other relevant evidence.

Examples

We can now use Agda to compute the non-terminating reduction sequence given earlier. First, we show that the term `sucμ` is well typed:

```
⊢sucμ : ∅ ⊢ μ "x" ⇒ `suc ` "x" ⦂ `ℕ
⊢sucμ = ⊢μ (⊢suc (⊢` ∋x))
  where
  ∋x = Z
```

To show the first three steps of the infinite reduction sequence, we evaluate with three steps worth of gas:

```
_ : eval (gas 3) ⊢sucμ ≡
  steps
    (μ "x" ⇒ `suc ` "x"
    —→⟨ β-μ ⟩
      `suc (μ "x" ⇒ `suc ` "x")
    —→⟨ ξ-suc β-μ ⟩
      `suc (`suc (μ "x" ⇒ `suc ` "x"))
    —→⟨ ξ-suc (ξ-suc β-μ) ⟩
      `suc (`suc (`suc (μ "x" ⇒ `suc ` "x")))
    ∎)
    out-of-gas
_ = refl
```

Similarly, we can use Agda to compute the reduction sequences given in the previous chapter. We start with the Church numeral two applied to successor and zero. Supplying 100 steps of gas is more than enough:

```
_ : eval (gas 100) (⊢twoᶜ · ⊢sucᶜ · ⊢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 `twoᶜ` and `sucᶜ` in place of their expansions.

Next, we show two plus two is four:

```
_ : eval (gas 100) ⊢2+2 ≡
  steps
    ((μ "+" ⇒
        (ƛ "m" ⇒
          (ƛ "n" ⇒
            case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
            ])))
      · `suc (`suc `zero)
      · `suc (`suc `zero)
    —→⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
      (ƛ "m" ⇒
        (ƛ "n" ⇒
          case ` "m" [zero⇒ ` "n" |suc "m" ⇒
          `suc
```

```
   ((μ "+" ⇒
     (ƛ "m" ⇒
      (ƛ "n" ⇒
       case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
       ])))
    · ` "m"
    · ` "n")
   ]))
 · `suc (`suc `zero)
 · `suc (`suc `zero)
—→⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
 (ƛ "n" ⇒
  case `suc (`suc `zero) [zero⇒ ` "n" |suc "m" ⇒
  `suc
  ((μ "+" ⇒
    (ƛ "m" ⇒
     (ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
      ])))
   · ` "m"
   · ` "n")
  ])
 · `suc (`suc `zero)
—→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
 case `suc (`suc `zero) [zero⇒ `suc (`suc `zero) |suc "m" ⇒
 `suc
 ((μ "+" ⇒
   (ƛ "m" ⇒
    (ƛ "n" ⇒
     case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
     ])))
  · ` "m"
  · `suc (`suc `zero))
 ]
—→⟨ β-suc (V-suc V-zero) ⟩
 `suc
 ((μ "+" ⇒
   (ƛ "m" ⇒
    (ƛ "n" ⇒
     case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
     ])))
  · `suc `zero
  · `suc (`suc `zero))
—→⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
 `suc
 ((ƛ "m" ⇒
   (ƛ "n" ⇒
    case ` "m" [zero⇒ ` "n" |suc "m" ⇒
    `suc
    ((μ "+" ⇒
      (ƛ "m" ⇒
       (ƛ "n" ⇒
        case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
        ])))
     · ` "m"
     · ` "n")
    ]))
  · `suc `zero
  · `suc (`suc `zero))
```

```
—→⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
 `suc
 ((ƛ "n" ⇒
   case `suc `zero [zero⇒ ` "n" |suc "m" ⇒
   `suc
   ((μ "+" ⇒
     (ƛ "m" ⇒
      (ƛ "n" ⇒
       case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
       ])))
    · ` "m"
    · ` "n")
   ])
  · `suc (`suc `zero))
—→⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
 `suc
 case `suc `zero [zero⇒ `suc (`suc `zero) |suc "m" ⇒
 `suc
 ((μ "+" ⇒
   (ƛ "m" ⇒
    (ƛ "n" ⇒
     case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
     ])))
  · ` "m"
  · `suc (`suc `zero))
 ]
—→⟨ ξ-suc (β-suc V-zero) ⟩
 `suc
 (`suc
  ((μ "+" ⇒
    (ƛ "m" ⇒
     (ƛ "n" ⇒
      case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
      ])))
   · `zero
   · `suc (`suc `zero)))
—→⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
 `suc
 (`suc
  ((ƛ "m" ⇒
    (ƛ "n" ⇒
     case ` "m" [zero⇒ ` "n" |suc "m" ⇒
     `suc
     ((μ "+" ⇒
       (ƛ "m" ⇒
        (ƛ "n" ⇒
         case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
         ])))
      · ` "m"
      · ` "n")
     ]))
   · `zero
   · `suc (`suc `zero)))
—→⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
 `suc
 (`suc
  ((ƛ "n" ⇒
    case `zero [zero⇒ ` "n" |suc "m" ⇒
    `suc
```

```
            ((μ "+" ⇒
                (ƛ "m" ⇒
                  (ƛ "n" ⇒
                    case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
                    ])))
              · ` "m"
              · ` "n")
            ])
          · `suc (`suc `zero)))
    —→⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
      `suc
      (`suc
        case `zero [zero⇒ `suc (`suc `zero) |suc "m" ⇒
        `suc
        ((μ "+" ⇒
            (ƛ "m" ⇒
              (ƛ "n" ⇒
                case ` "m" [zero⇒ ` "n" |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n")
                ])))
          · ` "m"
          · `suc (`suc `zero))
        ])
    —→⟨ ξ-suc (ξ-suc β-zero) ⟩
      `suc (`suc (`suc (`suc `zero)))
    ∎)
    (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

Again, the derivation in the previous chapter was derived by editing the above.

Similarly, we can evaluate the corresponding term for Church numerals:

```
_ : eval (gas 100) ⊢2+2ᶜ ≡
  steps
    ((ƛ "m" ⇒
        (ƛ "n" ⇒
          (ƛ "s" ⇒ (ƛ "z" ⇒ ` "m" · ` "s" · (` "n" · ` "s" · ` "z")))))
      · (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")))
      · (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")))
      · (ƛ "n" ⇒ `suc ` "n")
      · `zero
    —→⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
      (ƛ "n" ⇒
        (ƛ "s" ⇒
          (ƛ "z" ⇒
            (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · ` "s" ·
            (` "n" · ` "s" · ` "z"))))
      · (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")))
      · (ƛ "n" ⇒ `suc ` "n")
      · `zero
    —→⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
      (ƛ "s" ⇒
        (ƛ "z" ⇒
          (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · ` "s" ·
          ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · ` "s" · ` "z")))
      · (ƛ "n" ⇒ `suc ` "n")
      · `zero
    —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
      (ƛ "z" ⇒
    (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
    ·
    ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
     · ` "z"))
   · `zero
  —→⟨ β-ƛ V-zero ⟩
   (ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
   ·
   ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
    · `zero)
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
   (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
   ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
    · `zero)
  —→⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
   (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
   ((ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
    `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
   (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
   ((ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · `zero))
  —→⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
   (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
   ((ƛ "n" ⇒ `suc ` "n") · `suc `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
   (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
   `suc (`suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
   (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · `suc (`suc `zero))
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
   (ƛ "n" ⇒ `suc ` "n") · `suc (`suc (`suc `zero))
  —→⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
   `suc (`suc (`suc (`suc `zero)))
  ∎)
  (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

And again, the example in the previous section was derived by editing the above.

Exercise `mul-eval` **(recommended)**

Using the evaluator, confirm that two times two is four.

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

Exercise: `progress-preservation` **(practice)**

Without peeking at their statements above, write down the progress and preservation theorems for the simply typed lambda-calculus.

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

Exercise `subject_expansion` (practice)

We say that `M` *reduces* to `N` if `M —→ N`, but we can also describe the same situation by saying that `N` *expands* to `M`. The preservation property is sometimes called *subject reduction*. Its opposite is *subject expansion*, which holds if `M —→ N` and `∅ ⊢ N ⦂ A` imply `∅ ⊢ M ⦂ A`. Find two counter-examples to subject expansion, one with case expressions and one not involving case expressions.

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

Well-typed terms don't get stuck

A term is *normal* if it cannot reduce:

```
Normal : Term → Set
Normal M = ∀ {N} → ¬ (M —→ N)
```

A term is *stuck* if it is normal yet not a value:

```
Stuck : Term → Set
Stuck M = Normal M × ¬ Value M
```

Using progress, it is easy to show that no well-typed term is stuck:

```
postulate
  unstuck : ∀ {M A}
    → ∅ ⊢ M ⦂ A
      -----------
    → ¬ (Stuck M)
```

Using preservation, it is easy to show that after any number of steps, a well-typed term remains well typed:

```
postulate
  preserves : ∀ {M N A}
    → ∅ ⊢ M ⦂ A
    → M —↠ N
      ---------
    → ∅ ⊢ N ⦂ A
```

An easy consequence is that starting from a well-typed term, taking any number of reduction steps leads to a term that is not stuck:

```
postulate
  wttdgs : ∀ {M N A}
    → ∅ ⊢ M ⦂ A
    → M —↠ N
      -----------
    → ¬ (Stuck N)
```

Felleisen and Wright, who introduced proofs via progress and preservation, summarised this result with the slogan *well-typed terms don't get stuck*. (They were referring to earlier work by Robin

Milner, who used denotational rather than operational semantics. He introduced `wrong` as the denotation of a term with a type error, and showed *well-typed terms don't go wrong*.)

Exercise `stuck` (practice)

Give an example of an ill-typed term that does get stuck.

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

Exercise `unstuck` (recommended)

Provide proofs of the three postulates, `unstuck`, `preserves`, and `wttdgs` above.

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

Reduction is deterministic

When we introduced reduction, we claimed it was deterministic. For completeness, we present a formal proof here.

Our proof will need a variant of congruence to deal with functions of four arguments (to deal with `case_[zero⇒_|suc_⇒_]`). It is exactly analogous to `cong` and `cong₂` as defined previously:

```
cong₄ : ∀ {A B C D E : Set} (f : A → B → C → D → E)
  {s w : A} {t x : B} {u y : C} {v z : D}
  → s ≡ w → t ≡ x → u ≡ y → v ≡ z → f s t u v ≡ f w x y z
cong₄ f refl refl refl refl = refl
```

It is now straightforward to show that reduction is deterministic:

```
det : ∀ {M M′ M″}
  → (M —→ M′)
  → (M —→ M″)
    --------
  → M′ ≡ M″
det (ξ-·₁ L—→L′)   (ξ-·₁ L—→L″)     = cong₂ _·_ (det L—→L′ L—→L″) refl
det (ξ-·₁ L—→L′)   (ξ-·₂ VL M—→M″) = ⊥-elim (V¬—→ VL L—→L′)
det (ξ-·₁ L—→L′)   (β-ƛ _)          = ⊥-elim (V¬—→ V-ƛ L—→L′)
det (ξ-·₂ VL _)    (ξ-·₁ L—→L″)     = ⊥-elim (V¬—→ VL L—→L″)
det (ξ-·₂ _ M—→M′) (ξ-·₂ _ M—→M″)  = cong₂ _·_ refl (det M—→M′ M—→M″)
det (ξ-·₂ _ M—→M′) (β-ƛ VM)         = ⊥-elim (V¬—→ VM M—→M′)
det (β-ƛ _)        (ξ-·₁ L—→L″)     = ⊥-elim (V¬—→ V-ƛ L—→L″)
det (β-ƛ VM)       (ξ-·₂ _ M—→M″)   = ⊥-elim (V¬—→ VM M—→M″)
det (β-ƛ _)        (β-ƛ _)          = refl
det (ξ-suc M—→M′) (ξ-suc M—→M″)    = cong `suc_ (det M—→M′ M—→M″)
det (ξ-case L—→L′) (ξ-case L—→L″) = cong₄ case_[zero⇒_|suc_⇒_]
                                        (det L—→L′ L—→L″) refl refl refl
det (ξ-case L—→L′) β-zero          = ⊥-elim (V¬—→ V-zero L—→L′)
det (ξ-case L—→L′) (β-suc VL)      = ⊥-elim (V¬—→ (V-suc VL) L—→L′)
det β-zero         (ξ-case M—→M″)  = ⊥-elim (V¬—→ V-zero M—→M″)
det β-zero       β-zero          = refl
det (β-suc VL)   (ξ-case L—→L″)  = ⊥-elim (V¬—→ (V-suc VL) L—→L″)
det (β-suc _)    (β-suc _)       = refl
det β-μ          β-μ             = refl
```

The proof is by induction over possible reductions. We consider three typical cases:

- Two instances of `ξ-·₁`:

  ```
  L —→ L′                    L —→ L″
  --------------- ξ-·₁       --------------- ξ-·₁
  L · M —→ L′ · M            L · M —→ L″ · M
  ```

  By induction we have `L′ ≡ L″`, and hence by congruence `L′ · M ≡ L″ · M`.

- An instance of `ξ-·₁` and an instance of `ξ-·₂`:

  ```
                             Value L
  L —→ L′                    M —→ M″
  --------------- ξ-·₁       --------------- ξ-·₂
  L · M —→ L′ · M            L · M —→ L · M″
  ```

  The rule on the left requires `L` to reduce, but the rule on the right requires `L` to be a value. This is a contradiction since values do not reduce. If the value constraint was removed from `ξ-·₂`, or from one of the other reduction rules, then determinism would no longer hold.

- Two instances of `β-ƛ`:

  ```
  Value V                              Value V
  ------------------------------ β-ƛ   ------------------------------ β-ƛ
  (ƛ x ⇒ N) · V —→ N [ x := V ]        (ƛ x ⇒ N) · V —→ N [ x := V ]
  ```

  Since the left-hand sides are identical, the right-hand sides are also identical. The formal proof simply invokes `refl`.

Five of the 18 lines in the above proof are redundant, e.g., the case when one rule is `ξ-·₁` and the other is `ξ-·₂` is considered twice, once with `ξ-·₁` first and `ξ-·₂` second, and the other time with the two swapped. What we might like to do is delete the redundant lines and add

```
det M—→M′ M—→M″ = sym (det M—→M″ M—→M′)
```

to the bottom of the proof. But this does not work: the termination checker complains, because the arguments have merely switched order and neither is smaller.

Quiz

Suppose we add a new term `zap` with the following reduction rule

```
-------- β-zap
M —→ zap
```

and the following typing rule:

```
----------- ⊢zap
Γ ⊢ zap ⦂ A
```

Which of the following properties remain true in the presence of these rules? For each property, write either “remains true” or “becomes false.” If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Quiz

Suppose instead that we add a new term `foo` with the following reduction rules:

```
------------------ β-foo₁
(λ x ⇒ ` x) —→ foo

----------- β-foo₂
foo —→ zero
```

Which of the following properties remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Quiz

Suppose instead that we remove the rule `ξ·₁` from the step relation. Which of the following properties remain true in the absence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Quiz

We can enumerate all the computable function from naturals to naturals, by writing out all programs of type `` `ℕ ⇒ `ℕ `` in lexical order. Write `fᵢ` for the `i`’th function in this list.

Say we add a typing rule that applies the above enumeration to interpret a natural as a function from naturals to naturals:

```
Γ ⊢ L ⦂ `ℕ
Γ ⊢ M ⦂ `ℕ
-------------- _·ℕ_
Γ ⊢ L · M ⦂ `ℕ
```

And that we add the corresponding reduction rule:

```
fᵢ(m) ⟶ n
---------- δ
i · m ⟶ n
```

Which of the following properties remain true in the presence of this rule? For each one, write either “remains true” or else “becomes false.” If a property becomes false, give a counterexample:

- Determinism of `step`
- Progress
- Preservation

Are all properties preserved in this case? Are there any other alterations we would wish to make to the system?

Unicode

This chapter uses the following unicode:

```
ƛ  U+019B  LATIN SMALL LETTER LAMBDA WITH STROKE (\Gl-)
Δ  U+0394  GREEK CAPITAL LETTER DELTA (\GD or \Delta)
β  U+03B2  GREEK SMALL LETTER BETA (\Gb or \beta)
δ  U+03B4  GREEK SMALL LETTER DELTA (\Gd or \delta)
μ  U+03BC  GREEK SMALL LETTER MU (\Gm or \mu)
ξ  U+03BE  GREEK SMALL LETTER XI (\Gx or \xi)
ρ  U+03B4  GREEK SMALL LETTER RHO (\Gr or \rho)
ᵢ  U+1D62  LATIN SUBSCRIPT SMALL LETTER I (\_i)
ᶜ  U+1D9C  MODIFIER LETTER SMALL C (\^c)
–  U+2013  EM DASH (\em)
₄  U+2084  SUBSCRIPT FOUR (\_4)
↠  U+21A0  RIGHTWARDS TWO HEADED ARROW (\rr-)
⇒  U+21D2  RIGHTWARDS DOUBLE ARROW (\=>)
∅  U+2205  EMPTY SET (\0)
∋  U+220B  CONTAINS AS MEMBER (\ni)
≟  U+225F  QUESTIONED EQUAL TO (\?=)
⊢  U+22A2  RIGHT TACK (\vdash or \|-)
⦂  U+2982  Z NOTATION TYPE COLON (\:)
```

Chapter 13

DeBruijn: Intrinsically-typed de Bruijn representation

```
module plfa.part2.DeBruijn where
```

The previous two chapters introduced lambda calculus, with a formalisation based on named variables, and terms defined separately from types. We began with that approach because it is traditional, but it is not the one we recommend. This chapter presents an alternative approach, where named variables are replaced by de Bruijn indices and terms are indexed by their types. Our new presentation is more compact, using substantially fewer lines of code to cover the same ground.

There are two fundamental approaches to typed lambda calculi. One approach, followed in the last two chapters, is to first define terms and then define types. Terms exist independent of types, and may have types assigned to them by separate typing rules. Another approach, followed in this chapter, is to first define types and then define terms. Terms and type rules are intertwined, and it makes no sense to talk of a term without a type. The two approaches are sometimes called *Curry style* and *Church style*. Following Reynolds, we will refer to them as *extrinsic* and *intrinsic*.

The particular representation described here was first proposed by Thorsten Altenkirch and Bernhard Reus. The formalisation of renaming and substitution we use is due to Conor McBride. Related work has been carried out by James Chapman, James McKinna, and many others.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _<_; _≤?_; z≤n; s≤s)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Decidable using (True; toWitness)
```

Introduction

There is a close correspondence between the structure of a term and the structure of the derivation showing that it is well typed. For example, here is the term for the Church numeral two:

```
twoᶜ : Term
twoᶜ = ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")
```

And here is its corresponding type derivation:

```
⊢twoᶜ : ∀ {A} → ∅ ⊢ twoᶜ ⦂ Ch A
⊢twoᶜ = ⊢ƛ (⊢ƛ (⊢` ∋s · (⊢` ∋s · ⊢` ∋z)))
  where
  ∋s = S ("s" ≠ "z") Z
  ∋z = Z
```

(These are both taken from Chapter Lambda and you can see the corresponding derivation tree written out in full here.) The two definitions are in close correspondence, where:

- `` `_ `` corresponds to `` ⊢` ``
- `ƛ_⇒_` corresponds to `⊢ƛ`
- `_·_` corresponds to `_·_`

Further, if we think of `Z` as zero and `S` as successor, then the lookup derivation for each variable corresponds to a number which tells us how many enclosing binding terms to count to find the binding of that variable. Here `"z"` corresponds to `Z` or zero and `"s"` corresponds to `S Z` or one. And, indeed, `"z"` is bound by the inner abstraction (count outward past zero abstractions) and `"s"` is bound by the outer abstraction (count outward past one abstraction).

In this chapter, we are going to exploit this correspondence, and introduce a new notation for terms that simultaneously represents the term and its type derivation. Now we will write the following:

```
twoᶜ  :  ∅ ⊢ Ch `ℕ
twoᶜ  =  ƛ ƛ (# 1 · (# 1 · # 0))
```

A variable is represented by a natural number (written with `Z` and `S`, and abbreviated in the usual way), and tells us how many enclosing binding terms to count to find the binding of that variable. Thus, `# 0` is bound at the inner `ƛ`, and `# 1` at the outer `ƛ`.

Replacing variables by numbers in this way is called *de Bruijn representation*, and the numbers themselves are called *de Bruijn indices*, after the Dutch mathematician Nicolaas Govert (Dick) de Bruijn (1918—2012), a pioneer in the creation of proof assistants. One advantage of replacing named variables with de Bruijn indices is that each term now has a unique representation, rather than being represented by the equivalence class of terms under alpha renaming.

The other important feature of our chosen representation is that it is *intrinsically typed*. In the previous two chapters, the definition of terms and the definition of types are completely separate. All terms have type `Term`, and nothing in Agda prevents one from writing a nonsense term such as `` `zero · `suc `zero `` which has no type. Such terms that exist independent of types are sometimes called *preterms* or *raw terms*. Here we are going to replace the type `Term` of raw terms by the type `Γ ⊢ A` of intrinsically-typed terms which in context `Γ` have type `A`.

While these two choices fit well, they are independent. One can use de Bruijn indices in raw terms, or have intrinsically-typed terms with names. In Chapter Untyped, we will introduce terms with de Bruijn indices that are intrinsically scoped but not typed.

A second example

De Bruijn indices can be tricky to get the hang of, so before proceeding further let’s consider a second example. Here is the term that adds two naturals:

```
plus : Term
plus = μ "+" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
         case ` "m"
           [zero⇒ ` "n"
           |suc "m" ⇒ `suc (` "+" · ` "m" · ` "n") ]
```

Note variable `"m"` is bound twice, once in a lambda abstraction and once in the successor branch of the case. Any appearance of `"m"` in the successor branch must refer to the latter binding, due to shadowing.

Here is its corresponding type derivation:

```
⊢plus : ∅ ⊢ plus ⦂ `ℕ ⇒ `ℕ ⇒ `ℕ
⊢plus = ⊢μ (⊢ƛ (⊢ƛ (⊢case (⊢` ∋m) (⊢` ∋n)
         (⊢suc (⊢` ∋+ · ⊢` ∋m′ · ⊢` ∋n′)))))
  where
  ∋+  = (S ("+" ≠ "m") (S ("+" ≠ "n") (S ("+" ≠ "m") Z)))
  ∋m  = (S ("m" ≠ "n") Z)
  ∋n  = Z
  ∋m′ = Z
  ∋n′ = (S ("n" ≠ "m") Z)
```

The two definitions are in close correspondence, where in addition to the previous correspondences we have:

- `` `zero `` corresponds to `⊢zero`
- `` `suc_ `` corresponds to `⊢suc`
- `case_[zero⇒_|suc_⇒_]` corresponds to `⊢case`
- `μ_⇒_` corresponds to `⊢μ`

Note the two lookup judgments `∋m` and `∋m′` refer to two different bindings of variables named `"m"`. In contrast, the two judgments `∋n` and `∋n′` both refer to the same binding of `"n"` but accessed in different contexts, the first where `"n"` is the last binding in the context, and the second after `"m"` is bound in the successor branch of the case.

Here is the term and its type derivation in the notation of this chapter:

```
plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = μ ƛ ƛ case (# 1) (# 0) (`suc (# 3 · # 0 · # 1))
```

Reading from left to right, each de Bruijn index corresponds to a lookup derivation:

- `# 1` corresponds to `∋m`
- `# 0` corresponds to `∋n`
- `# 3` corresponds to `∋+`
- `# 0` corresponds to `∋m′`
- `# 1` corresponds to `∋n′`

The de Bruijn index counts the number of `S` constructs in the corresponding lookup derivation. Variable `"n"` bound in the inner abstraction is referred to as `# 0` in the zero branch of the case but as `# 1` in the successor branch of the case, because of the intervening binding. Variable `"m"` bound in the lambda abstraction is referred to by the first `# 1` in the code, while variable `"m"` bound in the successor branch of the case is referred to by the second `# 0`. There is no shadowing: with variable names, there is no way to refer to the former binding in the scope of the latter, but with de Bruijn indices it could be referred to as `# 2`.

Order of presentation

In the current chapter, the use of intrinsically-typed terms necessitates that we cannot introduce operations such as substitution or reduction without also showing that they preserve types. Hence, the order of presentation must change.

The syntax of terms now incorporates their typing rules, and the definition of values now incorporates the Canonical Forms lemma. The definition of substitution is somewhat more involved, but incorporates the trickiest part of the previous proof, the lemma establishing that substitution preserves types. The definition of reduction incorporates preservation, which no longer requires a separate proof.

Syntax

We now begin our formal development.

First, we get all our infix declarations out of the way. We list separately operators for judgments, types, and terms:

```
infix 4 _⊢_
infix 4 _∋_
infixl 5 _,_

infixr 7 _⇒_

infix 5 ƛ_
infix 5 μ_
infixl 7 _·_
infix 8 `suc_
infix 9 `_
infix 9 S_
infix 9 #_
```

Since terms are intrinsically typed, we must define types and contexts before terms.

Types

As before, we have just two types, functions and naturals. The formal definition is unchanged:

```
data Type : Set where
  _⇒_ : Type → Type → Type
  `ℕ : Type
```

Contexts

Contexts are as before, but we drop the names. Contexts are formalised as follows:

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

A context is just a list of types, with the type of the most recently bound variable on the right. As before, we let `Γ` and `Δ` range over contexts. We write `∅` for the empty context, and `Γ , A` for the context `Γ` extended by type `A`. For example

```
_ : Context
_ = ∅ , `ℕ ⇒ `ℕ , `ℕ
```

is a context with two variables in scope, where the outer bound one has type `` `ℕ ⇒ `ℕ ``, and the inner bound one has type `` `ℕ ``.

Variables and the lookup judgment

Intrinsically-typed variables correspond to the lookup judgment. They are represented by de Bruijn indices, and hence also correspond to natural numbers. We write

```
Γ ∋ A
```

for variables which in context `Γ` have type `A`. The lookup judgement is formalised by a datatype indexed by a context and a type. It looks exactly like the old lookup judgment, but with all variable names dropped:

```
data _∋_ : Context → Type → Set where

  Z : ∀ {Γ A}
      ---------
    → Γ , A ∋ A

  S_ : ∀ {Γ A B}
    → Γ ∋ A
      ---------
    → Γ , B ∋ A
```

Constructor `S` no longer requires an additional parameter, since without names shadowing is no longer an issue. Now constructors `Z` and `S` correspond even more closely to the constructors `here` and `there` for the element-of relation `_∈_` on lists, as well as to constructors `zero` and `suc` for natural numbers.

For example, consider the following old-style lookup judgments:

- `` ∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "z" ⦂ `ℕ ``
- `` ∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ∋ "s" ⦂ `ℕ ⇒ `ℕ ``

They correspond to the following intrinsically-typed variables:

```
_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ∋ `ℕ
_ = Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ∋ `ℕ ⇒ `ℕ
_ = S Z
```

In the given context, `"z"` is represented by `Z` (as the most recently bound variable), and `"s"` by `S Z` (as the next most recently bound variable).

Terms and the typing judgment

Intrinsically-typed terms correspond to the typing judgment. We write

```
Γ ⊢ A
```

for terms which in context `Γ` have type `A`. The judgement is formalised by a datatype indexed by a context and a type. It looks exactly like the old typing judgment, but with all terms and variable names dropped:

```
data _⊢_ : Context → Type → Set where

  `_ : ∀ {Γ A}
    → Γ ∋ A
      -----
    → Γ ⊢ A

  ƛ_  : ∀ {Γ A B}
    → Γ , A ⊢ B
      ---------
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
      ---------
    → Γ ⊢ B

  `zero : ∀ {Γ}
      ---------
    → Γ ⊢ `ℕ

  `suc_ : ∀ {Γ}
    → Γ ⊢ `ℕ
      ------
    → Γ ⊢ `ℕ

  case : ∀ {Γ A}
    → Γ ⊢ `ℕ
    → Γ ⊢ A
    → Γ , `ℕ ⊢ A
      ----------
    → Γ ⊢ A

  μ_ : ∀ {Γ A}
    → Γ , A ⊢ A
      ---------
    → Γ ⊢ A
```

The definition exploits the close correspondence between the structure of terms and the structure of a derivation showing that it is well typed: now we use the derivation *as* the term.

For example, consider the following old-style typing judgments:

- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" ⦂ `ℕ ⇒ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" · ` "z" ⦂ `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ , "z" ⦂ `ℕ ⊢ ` "s" · (` "s" · ` "z") ⦂  `ℕ`
- `∅ , "s" ⦂ `ℕ ⇒ `ℕ ⊢ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) ⦂ `ℕ ⇒ `ℕ`
- `∅ ⊢ ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · ` "z")) ⦂ (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ`

They correspond to the following intrinsically-typed terms:

```
_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ
_ = ` Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ ⇒ `ℕ
_ = ` S Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ
_ = ` S Z · ` Z

_ : ∅ , `ℕ ⇒ `ℕ , `ℕ ⊢ `ℕ
_ = ` S Z · (` S Z · ` Z)

_ : ∅ , `ℕ ⇒ `ℕ ⊢ `ℕ ⇒ `ℕ
_ = ƛ (` S Z · (` S Z · ` Z))

_ : ∅ ⊢ (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ
_ = ƛ ƛ (` S Z · (` S Z · ` Z))
```

The final term represents the Church numeral two.

Abbreviating de Bruijn indices

We define a helper function that computes the length of a context, which will be useful in making sure an index is within context bounds:

```
length : Context → ℕ
length ∅        = zero
length (Γ , _) = suc (length Γ)
```

We can use a natural number to select a type from a context:

```
lookup : {Γ : Context} → {n : ℕ} → (p : n < length Γ) → Type
lookup {(_ , A)} {zero} (s≤s z≤n)  = A
lookup {(Γ , _)} {(suc n)} (s≤s p) = lookup p
```

We intend to apply the function only when the natural is shorter than the length of the context, which is witnessed by `p`.

Given the above, we can convert a natural to a corresponding de Bruijn index, looking up its type in the context:

```
count : ∀ {Γ} → {n : ℕ} → (p : n < length Γ) → Γ ∋ lookup p
count {_ , _} {zero} (s≤s z≤n)  = Z
count {Γ , _} {(suc n)} (s≤s p) = S (count p)
```

We can then introduce a convenient abbreviation for variables:

```
#_ : ∀ {Γ}
  → (n : ℕ)
  → {n∈Γ : True (suc n ≤? length Γ)}
    --------------------------------
  → Γ ⊢ lookup (toWitness n∈Γ)
#_ n {n∈Γ} = ` count (toWitness n∈Γ)
```

Function `#_` takes an implicit argument `n∈Γ` that provides evidence for `n` to be within the context's bounds. Recall that `True`, `_≤?_` and `toWitness` are defined in Chapter Decidable. The type of `n∈Γ` guards against invoking `#_` on an `n` that is out of context bounds. Finally, in the return type `n∈Γ` is converted to a witness that `n` is within the bounds.

With this abbreviation, we can rewrite the Church numeral two more compactly:

```
_ : ∅ ⊢ (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ
_ = ƛ ƛ (# 1 · (# 1 · # 0))
```

Test examples

We repeat the test examples from Chapter Lambda. You can find them here for comparison.

First, computing two plus two on naturals:

```
two : ∀ {Γ} → Γ ⊢ `ℕ
two = `suc `suc `zero

plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = μ ƛ ƛ (case (# 1) (# 0) (`suc (# 3 · # 0 · # 1)))

2+2 : ∀ {Γ} → Γ ⊢ `ℕ
2+2 = plus · two · two
```

We generalise to arbitrary contexts because later we will give examples where `two` appears nested inside binders.

Next, computing two plus two on Church numerals:

```
Ch : Type → Type
Ch A = (A ⇒ A) ⇒ A ⇒ A

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:

```
_ : twoᶜ · sucᶜ · `zero {∅} —↠ `suc `suc `zero
_ =
  begin
    twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ (sucᶜ · (sucᶜ · # 0))) · `zero
  —→⟨ β-ƛ V-zero ⟩
    sucᶜ · (sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    sucᶜ · `suc `zero
  —→⟨ β-ƛ (V-suc V-zero) ⟩
    `suc (`suc `zero)
  ∎
```

As before, we need to supply an explicit context to `` `zero ``.

Next, a sample reduction demonstrating that two plus two is four:

```
_ : plus {∅} · two · two —↠ `suc `suc `suc `suc `zero
_ =
    plus · two · two
  —→⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
    (ƛ ƛ case (` S Z) (` Z) (`suc (plus · ` Z · ` S Z))) · two · two
  —→⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ case two (` Z) (`suc (plus · ` Z · ` S Z))) · two
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    case two two (`suc (plus · ` Z · two))
  —→⟨ β-suc (V-suc V-zero) ⟩
    `suc (plus · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
    `suc ((ƛ ƛ case (` S Z) (` Z) (`suc (plus · ` Z · ` S Z)))
      · `suc `zero · two)
  —→⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
    `suc ((ƛ case (`suc `zero) (` Z) (`suc (plus · ` Z · ` S Z))) · two)
  —→⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
    `suc (case (`suc `zero) (two) (`suc (plus · ` Z · two)))
  —→⟨ ξ-suc (β-suc V-zero) ⟩
    `suc (`suc (plus · `zero · two))
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
    `suc (`suc ((ƛ ƛ case (` S Z) (` Z) (`suc (plus · ` Z · ` S Z)))
      · `zero · two))
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
    `suc (`suc ((ƛ case `zero (` Z) (`suc (plus · ` Z · ` S Z))) · two))
  —→⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
    `suc (`suc (case `zero (two) (`suc (plus · ` Z · two))))
  —→⟨ ξ-suc (ξ-suc β-zero) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

And finally, a similar sample reduction for Church numerals:

```
_ : plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero —↠ `suc `suc `suc `suc `zero {∅}
_ =
  begin
    plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
    (ƛ ƛ ƛ twoᶜ · ` S Z · (` S S Z · ` S Z · ` Z)) · twoᶜ · sucᶜ · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ ƛ twoᶜ · ` S Z · (twoᶜ · ` S Z · ` Z)) · sucᶜ · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ twoᶜ · sucᶜ · (twoᶜ · sucᶜ · ` Z)) · `zero
  —→⟨ β-ƛ V-zero ⟩
    twoᶜ · sucᶜ · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · (twoᶜ · sucᶜ · `zero)
  —→⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · ((ƛ sucᶜ · (sucᶜ · ` Z)) · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · (sucᶜ · (sucᶜ · `zero))
  —→⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · (sucᶜ · `suc `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
    (ƛ sucᶜ · (sucᶜ · ` Z)) · `suc (`suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    sucᶜ · (sucᶜ · `suc (`suc `zero))
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    sucᶜ · `suc (`suc (`suc `zero))
  —→⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎
```

Values do not reduce

We have now completed all the definitions, which of necessity subsumed some of the propositions from the earlier development: Canonical Forms, Substitution preserves types, and Preservation. We now turn to proving the remaining results from the previous development.

Exercise `V¬—→` (practice)

Following the previous development, show values do not reduce, and its corollary, terms that reduce are not values.

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

Progress

As before, every term that is well typed and closed is either a value or takes a reduction step. The formulation of progress is just as before, but annotated with types:

```
data Progress {A} (M : ∅ ⊢ A) : Set where
```

```
  step : ∀ {N : ∅ ⊢ A}
    → M —→ N
      ----------
    → Progress M

  done :
      Value M
      ----------
    → Progress M
```

The statement and proof of progress is much as before, appropriately annotated. We no longer need to explicitly refer to the Canonical Forms lemma, since it is built-in to the definition of value:

```
progress : ∀ {A} → (M : ∅ ⊢ A) → Progress M
progress (` ())
progress (ƛ N)        = done V-ƛ
progress (L · M) with progress L
... | step L—→L′      = step (ξ-·₁ L—→L′)
... | done V-ƛ with progress M
... | step M—→M′      = step (ξ-·₂ V-ƛ M—→M′)
... | done VM         = step (β-ƛ VM)
progress (`zero)      = done V-zero
progress (`suc M) with progress M
... | step M—→M′      = step (ξ-suc M—→M′)
... | done VM         = done (V-suc VM)
progress (case L M N) with progress L
... | step L—→L′      = step (ξ-case L—→L′)
... | done V-zero     = step (β-zero)
... | done (V-suc VL) = step (β-suc VL)
progress (μ N)        = step (β-μ)
```

Evaluation

Before, we combined progress and preservation to evaluate a term. We can do much the same here, but we no longer need to explicitly refer to preservation, since it is built-in to the definition of reduction.

As previously, gas is specified by a natural number:

```
record Gas : Set where
  constructor gas
  field
    amount : ℕ
```

When our evaluator returns a term `N`, it will either give evidence that `N` is a value or indicate that it ran out of gas:

```
data Finished {Γ A} (N : Γ ⊢ A) : Set where

  done :
    Value N
    ----------
    → Finished N

  out-of-gas :
    ----------
```

```
   Finished N
```

Given a term `L` of type `A`, the evaluator will, for some `N`, return a reduction sequence from `L` to `N` and an indication of whether reduction finished:

```
data Steps {A} : ∅ ⊢ A → Set where

  steps : {L N : ∅ ⊢ A}
    → L —↠ N
    → Finished N
      ----------
    → Steps L
```

The evaluator takes gas and a term and returns the corresponding steps:

```
eval : ∀ {A}
  → Gas
  → (L : ∅ ⊢ A)
    -----------
  → Steps L
eval (gas zero) L    = steps (L ∎) out-of-gas
eval (gas (suc m)) L with progress L
... | done VL         = steps (L ∎) (done VL)
... | step {M} L—→M with eval (gas m) M
... | steps M—↠N fin = steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

The definition is a little simpler than previously, as we no longer need to invoke preservation.

Examples

We reiterate each of our previous examples. We re-define the term `sucμ` that loops forever:

```
sucμ : ∅ ⊢ `ℕ
sucμ = μ (`suc (# 0))
```

To compute the first three steps of the infinite reduction sequence, we evaluate with three steps worth of gas:

```
_ : eval (gas 3) sucμ ≡
  steps
    (μ `suc ` Z
    —→⟨ β-μ ⟩
      `suc (μ `suc ` Z)
    —→⟨ ξ-suc β-μ ⟩
      `suc (`suc (μ `suc ` Z))
    —→⟨ ξ-suc (ξ-suc β-μ) ⟩
      `suc (`suc (`suc (μ `suc ` Z)))
    ∎)
    out-of-gas
_ = refl
```

The Church numeral two applied to successor and zero:

```
_ : eval (gas 100) (twoᶜ · sucᶜ · `zero) ≡
  steps
   ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · `zero
   —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) · `zero
   —→⟨ β-ƛ V-zero ⟩
    (ƛ `suc ` Z) · ((ƛ `suc ` Z) · `zero)
   —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ `suc ` Z) · `suc `zero
   —→⟨ β-ƛ (V-suc V-zero) ⟩
    `suc (`suc `zero)
   ∎)
   (done (V-suc (V-suc V-zero)))
_ = refl
```

Two plus two is four:

```
_ : eval (gas 100) (plus · two · two) ≡
  steps
   ((μ
      (ƛ
        (ƛ
          case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
    · `suc (`suc `zero)
    · `suc (`suc `zero)
   —→⟨ ξ-·₁ (ξ-·₁ β-μ) ⟩
    (ƛ
      (ƛ
        case (` (S Z)) (` Z)
        (`suc
          ((μ
              (ƛ
                (ƛ
                  case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
            · ` Z
            · ` (S Z)))))
    · `suc (`suc `zero)
    · `suc (`suc `zero)
   —→⟨ ξ-·₁ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ
      case (`suc (`suc `zero)) (` Z)
      (`suc
        ((μ
            (ƛ
              (ƛ
                case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
          · ` Z
          · ` (S Z))))
    · `suc (`suc `zero)
   —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    case (`suc (`suc `zero)) (`suc (`suc `zero))
    (`suc
      ((μ
          (ƛ
            (ƛ
              case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
        · ` Z
        · `suc (`suc `zero)))
   —→⟨ β-suc (V-suc V-zero) ⟩
```

```
    `suc
    ((μ
        (ƛ
          (ƛ
            case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
      · `suc `zero
      · `suc (`suc `zero))
  —→⟨ ξ-suc (ξ-·₁ (ξ-·₁ β-μ)) ⟩
    `suc
    ((ƛ
        (ƛ
          case (` (S Z)) (` Z)
          (`suc
            ((μ
                (ƛ
                  (ƛ
                    case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
              · ` Z
              · ` (S Z)))))
      · `suc `zero
      · `suc (`suc `zero))
  —→⟨ ξ-suc (ξ-·₁ (β-ƛ (V-suc V-zero))) ⟩
    `suc
    ((ƛ
        case (`suc `zero) (` Z)
        (`suc
          ((μ
              (ƛ
                (ƛ
                  case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
            · ` Z
            · ` (S Z))))
      · `suc (`suc `zero))
  —→⟨ ξ-suc (β-ƛ (V-suc (V-suc V-zero))) ⟩
    `suc
    case (`suc `zero) (`suc (`suc `zero))
    (`suc
      ((μ
          (ƛ
            (ƛ
              case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
        · ` Z
        · `suc (`suc `zero)))
  —→⟨ ξ-suc (β-suc V-zero) ⟩
    `suc
    (`suc
      ((μ
          (ƛ
            (ƛ
              case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
        · `zero
        · `suc (`suc `zero)))
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (ξ-·₁ β-μ))) ⟩
    `suc
    (`suc
      ((ƛ
          (ƛ
            case (` (S Z)) (` Z)
            (`suc
        ((μ
          (ƛ
           (ƛ
            case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
         · ` Z
         · ` (S Z)))))
     · `zero
     · `suc (`suc `zero)))
  —→⟨ ξ-suc (ξ-suc (ξ-·₁ (β-ƛ V-zero))) ⟩
   `suc
   (`suc
    ((ƛ
      case `zero (` Z)
      (`suc
       ((μ
         (ƛ
          (ƛ
           case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
        · ` Z
        · ` (S Z))))
     · `suc (`suc `zero)))
  —→⟨ ξ-suc (ξ-suc (β-ƛ (V-suc (V-suc V-zero)))) ⟩
   `suc
   (`suc
    case `zero (`suc (`suc `zero))
    (`suc
     ((μ
       (ƛ
        (ƛ
         case (` (S Z)) (` Z) (`suc (` (S (S (S Z))) · ` Z · ` (S Z))))))
      · ` Z
      · `suc (`suc `zero))))
  —→⟨ ξ-suc (ξ-suc β-zero) ⟩
   `suc (`suc (`suc (`suc `zero)))
  ∎)
  (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

And the corresponding term for Church numerals:

```
_ : eval (gas 100) (plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero) ≡
 steps
  ((ƛ
    (ƛ
     (ƛ (ƛ ` (S (S (S Z))) · ` (S Z) · (` (S (S Z)) · ` (S Z) · ` Z)))))
   · (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z)))
   · (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z)))
   · (ƛ `suc ` Z)
   · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ))) ⟩
   (ƛ
    (ƛ
     (ƛ
      (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · ` (S Z) ·
      (` (S (S Z)) · ` (S Z) · ` Z))))
   · (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z)))
   · (ƛ `suc ` Z)
   · `zero
  —→⟨ ξ-·₁ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
```

```
    (ƛ
      (ƛ
        (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · ` (S Z) ·
        ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · ` (S Z) · ` Z)))
    · (ƛ `suc ` Z)
    · `zero
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ
      (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) ·
      ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · ` Z))
    · `zero
  —→⟨ β-ƛ V-zero ⟩
    (ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) ·
    ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · `zero)
  —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ (ƛ ` (S Z) · (` (S Z) · ` Z))) · (ƛ `suc ` Z) · `zero)
  —→⟨ ξ-·₂ V-ƛ (ξ-·₁ (β-ƛ V-ƛ)) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) · `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ `suc ` Z) · ((ƛ `suc ` Z) · `zero))
  —→⟨ ξ-·₂ V-ƛ (ξ-·₂ V-ƛ (β-ƛ V-zero)) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) ·
    ((ƛ `suc ` Z) · `suc `zero)
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc V-zero)) ⟩
    (ƛ (ƛ `suc ` Z) · ((ƛ `suc ` Z) · ` Z)) · `suc (`suc `zero)
  —→⟨ β-ƛ (V-suc (V-suc V-zero)) ⟩
    (ƛ `suc ` Z) · ((ƛ `suc ` Z) · `suc (`suc `zero))
  —→⟨ ξ-·₂ V-ƛ (β-ƛ (V-suc (V-suc V-zero))) ⟩
    (ƛ `suc ` Z) · `suc (`suc (`suc `zero))
  —→⟨ β-ƛ (V-suc (V-suc (V-suc V-zero))) ⟩
    `suc (`suc (`suc (`suc `zero)))
  ∎)
  (done (V-suc (V-suc (V-suc (V-suc V-zero)))))
_ = refl
```

We omit the proof that reduction is deterministic, since it is tedious and almost identical to the previous proof.

Exercise `mul-example` (recommended)

Using the evaluator, confirm that two times two is four.

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

Intrinsic typing is golden

Counting the lines of code is instructive. While this chapter covers the same formal development as the previous two chapters, it has much less code. Omitting all the examples, and all proofs that appear in Properties but not DeBruijn (such as the proof that reduction is deterministic), the number of lines of code is as follows:

```
Lambda                      216
Properties                  235
DeBruijn                    276
```

The relation between the two approaches approximates the golden ratio: extrinsically-typed terms require about 1.6 times as much code as intrinsically-typed.

Unicode

This chapter uses the following unicode:

```
σ  U+03C3  GREEK SMALL LETTER SIGMA (\Gs or \sigma)
₀  U+2080  SUBSCRIPT ZERO (\_0)
₃  U+20B3  SUBSCRIPT THREE (\_3)
₄  U+2084  SUBSCRIPT FOUR (\_4)
₅  U+2085  SUBSCRIPT FIVE (\_5)
₆  U+2086  SUBSCRIPT SIX (\_6)
₇  U+2087  SUBSCRIPT SEVEN (\_7)
≠  U+2260  NOT EQUAL TO (\=n)
```

Chapter 14

More: Additional constructs of simply-typed lambda calculus

```
module plfa.part2.More where
```

So far, we have focussed on a relatively minimal language, based on Plotkin's PCF, which supports functions, naturals, and fixpoints. In this chapter we extend our calculus to support the following:

- primitive numbers
- *let* bindings
- products
- an alternative formulation of products
- sums
- unit type
- an alternative formulation of unit type
- empty type
- lists

All of the data types should be familiar from Part I of this textbook. For *let* and the alternative formulations we show how they translate to other constructs in the calculus. Most of the description will be informal. We show how to formalise the first four constructs and leave the rest as an exercise for the reader.

Our informal descriptions will be in the style of Chapter Lambda, using extrinsically-typed terms, while our formalisation will be in the style of Chapter DeBruijn, using intrinsically-typed terms.

By now, explaining with symbols should be more concise, more precise, and easier to follow than explaining in prose. For each construct, we give syntax, typing, reductions, and an example. We also give translations where relevant; formally establishing the correctness of translations will be the subject of the next chapter.

Primitive numbers

We define a `Nat` type equivalent to the built-in natural number type with multiplication as a primitive operation on numbers:

Syntax

```
A, B, C ::= ...                    Types
  Nat                                primitive natural numbers

L, M, N ::= ...                    Terms
  con c                              constant
  L `* M                             multiplication

V, W ::= ...                       Values
  con c                              constant
```

Typing

The hypothesis of the `con` rule is unusual, in that it refers to a typing judgment of Agda rather than a typing judgment of the defined calculus:

```
c : ℕ
--------------- con
Γ ⊢ con c : Nat

Γ ⊢ L : Nat
Γ ⊢ M : Nat
---------------- _`*_
Γ ⊢ L `* M : Nat
```

Reduction

A rule that defines a primitive directly, such as the last rule below, is called a δ rule. Here the δ rule defines multiplication of primitive numbers in terms of multiplication of naturals as given by the Agda standard prelude:

```
L —→ L′
----------------- ξ-*₁
L `* M —→ L′ `* M

M —→ M′
----------------- ξ-*₂
V `* M —→ V `* M′

----------------------------- δ-*
con c `* con d —→ con (c * d)
```

Example

Here is a function to cube a primitive number:

```
cube : ∅ ⊢ Nat ⇒ Nat
cube = ƛ x ⇒ x `* x `* x
```

Let bindings

Let bindings affect only the syntax of terms; they introduce no new types or values:

Syntax

```
L, M, N ::= ...                     Terms
  `let x `= M `in N                   let
```

Typing

```
Γ ⊢ M ⦂ A
Γ , x ⦂ A ⊢ N ⦂ B
------------------------- `let
Γ ⊢ `let x `= M `in N ⦂ B
```

Reduction

```
M —→ M′
--------------------------------------- ξ-let
`let x `= M `in N —→ `let x `= M′ `in N

--------------------------------- β-let
`let x `= V `in N —→ N [ x := V ]
```

Example

Here is a function to raise a primitive number to the tenth power:

```
exp10 : ∅ ⊢ Nat ⇒ Nat
exp10 = ƛ x ⇒ `let x2  `= x  `* x  `in
              `let x4  `= x2 `* x2 `in
              `let x5  `= x4 `* x  `in
              x5 `* x5
```

Translation

We can translate each *let* term into an application of an abstraction:

```
(`let x `= M `in N) †  =  (ƛ x ⇒ (N †)) · (M †)
```

Here `M †` is the translation of term `M` from a calculus with the construct to a calculus without the construct.

Products

Syntax

```
A, B, C ::= ...                     Types
  A `× B                              product type

L, M, N ::= ...                     Terms
  `⟨ M , N ⟩                          pair
  `proj₁ L                            project first component
  `proj₂ L                            project second component

V, W ::= ...                        Values
  `⟨ V , W ⟩                          pair
```

Typing

```
Γ ⊢ M ⦂ A
Γ ⊢ N ⦂ B
----------------------- `⟨_,_⟩ or `×-I
Γ ⊢ `⟨ M , N ⟩ ⦂ A `× B

Γ ⊢ L ⦂ A `× B
---------------- `proj₁ or `×-E₁
Γ ⊢ `proj₁ L ⦂ A

Γ ⊢ L ⦂ A `× B
---------------- `proj₂ or `×-E₂
Γ ⊢ `proj₂ L ⦂ B
```

Reduction

```
M —→ M′
------------------------- ξ-⟨,⟩₁
`⟨ M , N ⟩ —→ `⟨ M′ , N ⟩

N —→ N′
------------------------- ξ-⟨,⟩₂
`⟨ V , N ⟩ —→ `⟨ V , N′ ⟩

L —→ L′
--------------------- ξ-proj₁
`proj₁ L —→ `proj₁ L′

L —→ L′
--------------------- ξ-proj₂
`proj₂ L —→ `proj₂ L′

---------------------- β-proj₁
`proj₁ `⟨ V , W ⟩ —→ V
```

```
---------------------- β-proj₂
`proj₂ `⟨ V , W ⟩ —→ W
```

Example

Here is a function to swap the components of a pair:

```
swap× : ∅ ⊢ A `× B ⇒ B `× A
swap× = ƛ z ⇒ `⟨ `proj₂ z , `proj₁ z ⟩
```

Alternative formulation of products

There is an alternative formulation of products, where in place of two ways to eliminate the type we have a case term that binds two variables. We repeat the syntax in full, but only give the new type and reduction rules:

Syntax

```
A, B, C ::= ...                    Types
  A `× B                             product type

L, M, N ::= ...                    Terms
  `⟨ M , N ⟩                         pair
  case× L [⟨ x , y ⟩⇒ M ]            case

V, W ::=                           Values
  `⟨ V , W ⟩                         pair
```

Typing

```
Γ ⊢ L ⦂ A `× B
Γ , x ⦂ A , y ⦂ B ⊢ N ⦂ C
------------------------------- case× or ×-E
Γ ⊢ case× L [⟨ x , y ⟩⇒ N ] ⦂ C
```

Reduction

```
L —→ L′
--------------------------------------------------- ξ-case×
case× L [⟨ x , y ⟩⇒ N ] —→ case× L′ [⟨ x , y ⟩⇒ N ]

--------------------------------------------------------- β-case×
case× `⟨ V , W ⟩ [⟨ x , y ⟩⇒ N ] —→ N [ x := V ][ y := W ]
```

Example

Here is a function to swap the components of a pair rewritten in the new notation:

```
swap×-case : ∅ ⊢ A `× B ⇒ B `× A
swap×-case = ƛ z ⇒ case× z
                     [⟨ x , y ⟩⇒ `⟨ y , x ⟩ ]
```

Translation

We can translate the alternative formulation into the one with projections:

```
  (case× L [⟨ x , y ⟩⇒ N ]) †
=
  `let z `= (L †) `in
  `let x `= `proj₁ z `in
  `let y `= `proj₂ z `in
  (N †)
```

Here `z` is a variable that does not appear free in `N`. We refer to such a variable as *fresh*.

One might think that we could instead use a more compact translation:

```
-- WRONG
  (case× L [⟨ x , y ⟩⇒ N ]) †
=
  (N †) [ x := `proj₁ (L †) ] [ y := `proj₂ (L †) ]
```

But this behaves differently. The first term always reduces `L` before `N`, and it computes `proj₁`` and proj₂ ``exactly once. The second term does not reduce `L` to a value before reducing `N`
and ‘`proj₂` many times or not at all.

We can also translate back the other way:

```
(`proj₁ L) ‡  =  case× (L ‡) [⟨ x , y ⟩⇒ x ]
(`proj₂ L) ‡  =  case× (L ‡) [⟨ x , y ⟩⇒ y ]
```

Sums

Syntax

```
A, B, C ::= ...                        Types
  A `⊎ B                                 sum type

L, M, N ::= ...                        Terms
  `inj₁ M                                inject first component
  `inj₂ N                                inject second component
  case⊎ L [inj₁ x ⇒ M |inj₂ y ⇒ N ]      case

V, W ::= ...                           Values
```

```
  `inj₁ V                             inject first component
  `inj₂ W                             inject second component
```

Typing

```
Γ ⊢ M ⦂ A
-------------------- `inj₁ or ⊎-I₁
Γ ⊢ `inj₁ M ⦂ A `⊎ B

Γ ⊢ N ⦂ B
-------------------- `inj₂ or ⊎-I₂
Γ ⊢ `inj₂ N ⦂ A `⊎ B

Γ ⊢ L ⦂ A `⊎ B
Γ , x ⦂ A ⊢ M ⦂ C
Γ , y ⦂ B ⊢ N ⦂ C
----------------------------------------- case⊎ or ⊎-E
Γ ⊢ case⊎ L [inj₁ x ⇒ M |inj₂ y ⇒ N ] ⦂ C
```

Reduction

```
M —→ M′
------------------- ξ-inj₁
`inj₁ M —→ `inj₁ M′

N —→ N′
------------------- ξ-inj₂
`inj₂ N —→ `inj₂ N′

L —→ L′
---------------------------------------------------------------------- ξ-case⊎
case⊎ L [inj₁ x ⇒ M |inj₂ y ⇒ N ] —→ case⊎ L′ [inj₁ x ⇒ M |inj₂ y ⇒ N ]

--------------------------------------------------------- β-inj₁
case⊎ (`inj₁ V) [inj₁ x ⇒ M |inj₂ y ⇒ N ] —→ M [ x := V ]

--------------------------------------------------------- β-inj₂
case⊎ (`inj₂ W) [inj₁ x ⇒ M |inj₂ y ⇒ N ] —→ N [ y := W ]
```

Example

Here is a function to swap the components of a sum:

```
swap⊎ : ∅ ⊢ A `⊎ B ⇒ B `⊎ A
swap⊎ = ƛ z ⇒ case⊎ z
                [inj₁ x ⇒ `inj₂ x
                |inj₂ y ⇒ `inj₁ y ]
```

Unit type

For the unit type, there is a way to introduce values of the type but no way to eliminate values of the type. There are no reduction rules.

Syntax

```
A, B, C ::= ...                     Types
  `⊤                                  unit type

L, M, N ::= ...                     Terms
  `tt                                 unit value

V, W ::= ...                        Values
  `tt                                 unit value
```

Typing

```
------------ `tt or ⊤-I
Γ ⊢ `tt ⦂ `⊤
```

Reduction

(none)

Example

Here is the isomorphism between `A` and `` A `× `⊤ ``:

```
to×⊤ : ∅ ⊢ A ⇒ A `× `⊤
to×⊤ = ƛ x ⇒ `⟨ x , `tt ⟩

from×⊤ : ∅ ⊢ A `× `⊤ ⇒ A
from×⊤ = ƛ z ⇒ `proj₁ z
```

Alternative formulation of unit type

There is an alternative formulation of the unit type, where in place of no way to eliminate the type we have a case term that binds zero variables. We repeat the syntax in full, but only give the new type and reduction rules:

Syntax

```
A, B, C ::= ...                     Types
  `⊤                                  unit type

L, M, N ::= ...                     Terms
  `tt                                 unit value
  `case⊤ L [tt⇒ N ]                   case

V, W ::= ...                        Values
  `tt                                 unit value
```

Typing

```
Γ ⊢ L ⦂ `⊤
Γ ⊢ M ⦂ A
------------------------ case⊤ or ⊤-E
Γ ⊢ case⊤ L [tt⇒ M ] ⦂ A
```

Reduction

```
L —→ L′
------------------------------------- ξ-case⊤
case⊤ L [tt⇒ M ] —→ case⊤ L′ [tt⇒ M ]

----------------------- β-case⊤
case⊤ `tt [tt⇒ M ] —→ M
```

Example

Here is half the isomorphism between `A` and ``A `× `⊤`` rewritten in the new notation:

```
from×⊤-case : ∅ ⊢ A `× `⊤ ⇒ A
from×⊤-case = ƛ z ⇒ case× z
                      [⟨ x , y ⟩⇒ case⊤ y
                                    [tt⇒ x ] ]
```

Translation

We can translate the alternative formulation into one without case:

```
(case⊤ L [tt⇒ M ]) †  =  `let z `= (L †) `in (M †)
```

Here `z` is a variable that does not appear free in `M`.

Empty type

For the empty type, there is a way to eliminate values of the type but no way to introduce values of the type. There are no values of the type and no β rule, but there is a ξ rule. The `case⊥` construct plays a role similar to `⊥-elim` in Agda:

Syntax

```
A, B, C ::= ...                     Types
  `⊥                                  empty type

L, M, N ::= ...                     Terms
  case⊥ L []                          case
```

Typing

```
Γ ⊢ L ⦂ `⊥
------------------ case⊥ or ⊥-E
Γ ⊢ case⊥ L [] ⦂ A
```

Reduction

```
L —→ L′
-------------------------- ξ-case⊥
case⊥ L [] —→ case⊥ L′ []
```

Example

Here is the isomorphism between `A` and `` A `⊎ `⊥ ``:

```
to⊎⊥ : ∅ ⊢ A ⇒ A `⊎ `⊥
to⊎⊥ = ƛ x ⇒ `inj₁ x

from⊎⊥ : ∅ ⊢ A `⊎ `⊥ ⇒ A
from⊎⊥ = ƛ z ⇒ case⊎ z
                 [inj₁ x ⇒ x
                 |inj₂ y ⇒ case⊥ y
                             [] ]
```

Lists

Syntax

```
A, B, C ::= ...                    Types
  `List A                            list type

L, M, N ::= ...                    Terms
  `[]                                nil
  M `∷ N                             cons
  caseL L [[]⇒ M | x ∷ y ⇒ N ]       case

V, W ::= ...                       Values
  `[]                                nil
  V `∷ W                             cons
```

Typing

```
----------------- `[] or List-I₁
Γ ⊢ `[] ⦂ `List A

Γ ⊢ M ⦂ A
Γ ⊢ N ⦂ `List A
-------------------- _`∷_ or List-I₂
Γ ⊢ M `∷ N ⦂ `List A

Γ ⊢ L ⦂ `List A
Γ ⊢ M ⦂ B
Γ , x ⦂ A , xs ⦂ `List A ⊢ N ⦂ B
-------------------------------------- caseL or List-E
Γ ⊢ caseL L [[]⇒ M | x ∷ xs ⇒ N ] ⦂ B
```

Reduction

```
M —→ M′
----------------- ξ-∷₁
M `∷ N —→ M′ `∷ N

N —→ N′
----------------- ξ-∷₂
V `∷ N —→ V `∷ N′

L —→ L′
--------------------------------------------------------------- ξ-caseL
caseL L [[]⇒ M | x ∷ xs ⇒ N ] —→ caseL L′ [[]⇒ M | x ∷ xs ⇒ N ]

------------------------------------ β-[]
caseL `[] [[]⇒ M | x ∷ xs ⇒ N ] —→ M

--------------------------------------------------------------- β-∷
caseL (V `∷ W) [[]⇒ M | x ∷ xs ⇒ N ] —→ N [ x := V ][ xs := W ]
```

Example

Here is the map function for lists:

```
mapL : ∅ ⊢ (A ⇒ B) ⇒ `List A ⇒ `List B
mapL = μ mL ⇒ ƛ f ⇒ ƛ xs ⇒
         caseL xs
           [[]⇒ `[]
           | x ∷ xs ⇒ f · x `∷ mL · f · xs ]
```

Formalisation

We now show how to formalise

- primitive numbers
- *let* bindings
- products
- an alternative formulation of products

and leave formalisation of the remaining constructs as an exercise.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _*_; _<_; _≤?_; z≤n; s≤s)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Decidable using (True; toWitness)
```

Syntax

```
infix 4 _⊢_
infix 4 _∋_
infixl 5 _,_

infixr 7 _⇒_
infixr 9 _`×_

infix 5 ƛ_
infix 5 μ_
infixl 7 _·_
infixl 8 _`*_
infix 8 `suc_
infix 9 `_
infix 9 S_
infix 9 #_
```

Types

```
data Type : Set where
  `ℕ   : Type
  _⇒_  : Type → Type → Type
  Nat  : Type
  _`×_ : Type → Type → Type
```

Contexts

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

Variables and the lookup judgment

```
data _∋_ : Context → Type → Set where

  Z : ∀ {Γ A}
      ---------
    → Γ , A ∋ A

  S_ : ∀ {Γ A B}
    → Γ ∋ B
      ---------
    → Γ , A ∋ B
```

Terms and the typing judgment

```
data _⊢_ : Context → Type → Set where

  -- variables

  `_ : ∀ {Γ A}
    → Γ ∋ A
      -----
    → Γ ⊢ A

  -- functions

  ƛ_  : ∀ {Γ A B}
    → Γ , A ⊢ B
      ---------
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
      ---------
    → Γ ⊢ B
```

```
-- naturals

`zero : ∀ {Γ}
    ------
  → Γ ⊢ `ℕ

`suc_ : ∀ {Γ}
  → Γ ⊢ `ℕ
    ------
  → Γ ⊢ `ℕ

case : ∀ {Γ A}
  → Γ ⊢ `ℕ
  → Γ ⊢ A
  → Γ , `ℕ ⊢ A
    -----
  → Γ ⊢ A

-- fixpoint

μ_ : ∀ {Γ A}
  → Γ , A ⊢ A
    ----------
  → Γ ⊢ A

-- primitive numbers

con : ∀ {Γ}
  → ℕ
    -------
  → Γ ⊢ Nat

_`*_ : ∀ {Γ}
  → Γ ⊢ Nat
  → Γ ⊢ Nat
    -------
  → Γ ⊢ Nat

-- let

`let : ∀ {Γ A B}
  → Γ ⊢ A
  → Γ , A ⊢ B
    ----------
  → Γ ⊢ B

-- products

`⟨_,_⟩ : ∀ {Γ A B}
  → Γ ⊢ A
  → Γ ⊢ B
    -----------
  → Γ ⊢ A `× B

`proj₁ : ∀ {Γ A B}
  → Γ ⊢ A `× B
    -----------
  → Γ ⊢ A

`proj₂ : ∀ {Γ A B}
  → Γ ⊢ A `× B
    -----------
```

```
   → Γ ⊢ B

  -- alternative formulation of products

  case× : ∀ {Γ A B C}
    → Γ ⊢ A `× B
    → Γ , A , B ⊢ C
      --------------
    → Γ ⊢ C
```

Abbreviating de Bruijn indices

```
length : Context → ℕ
length ∅        = zero
length (Γ , _) = suc (length Γ)

lookup : {Γ : Context} → {n : ℕ} → (p : n < length Γ) → Type
lookup {(_ , A)} {zero} (s≤s z≤n)  = A
lookup {(Γ , _)} {(suc n)} (s≤s p) = lookup p

count : ∀ {Γ} → {n : ℕ} → (p : n < length Γ) → Γ ∋ lookup p
count {_ , _} {zero} (s≤s z≤n)  =  Z
count {Γ , _} {(suc n)} (s≤s p) =  S (count p)

#_ : ∀ {Γ}
  → (n : ℕ)
  → {n∈Γ : True (suc n ≤? length Γ)}
    --------------------------------
  → Γ ⊢ lookup (toWitness n∈Γ)
#_ n {n∈Γ} = ` count (toWitness n∈Γ)
```

Renaming

```
ext : ∀ {Γ Δ}
  → (∀ {A}  → Γ ∋ A → Δ ∋ A)
    ---------------------------------
  → (∀ {A B} → Γ , A ∋ B → Δ , A ∋ B)
ext ρ Z      = Z
ext ρ (S x) = S (ρ x)

rename : ∀ {Γ Δ}
  → (∀ {A} → Γ ∋ A → Δ ∋ A)
    -----------------------
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
rename ρ (` x)          = ` (ρ x)
rename ρ (ƛ N)          = ƛ (rename (ext ρ) N)
rename ρ (L · M)        = (rename ρ L) · (rename ρ M)
rename ρ (`zero)        = `zero
rename ρ (`suc M)       = `suc (rename ρ M)
rename ρ (case L M N) = case (rename ρ L) (rename ρ M) (rename (ext ρ) N)
rename ρ (μ N)          = μ (rename (ext ρ) N)
rename ρ (con n)        = con n
rename ρ (M `* N)      = rename ρ M `* rename ρ N
rename ρ (`let M N)    = `let (rename ρ M) (rename (ext ρ) N)
rename ρ `⟨ M , N ⟩    = `⟨ rename ρ M , rename ρ N ⟩
rename ρ (`proj₁ L)    = `proj₁ (rename ρ L)
rename ρ (`proj₂ L)    = `proj₂ (rename ρ L)
rename ρ (case× L M)   = case× (rename ρ L) (rename (ext (ext ρ)) M)
```

Simultaneous Substitution

```
exts : ∀ {Γ Δ} → (∀ {A} → Γ ∋ A → Δ ⊢ A) → (∀ {A B} → Γ , A ∋ B → Δ , A ⊢ B)
exts σ Z      =  ` Z
exts σ (S x)  =  rename S_ (σ x)

subst : ∀ {Γ Δ} → (∀ {C} → Γ ∋ C → Δ ⊢ C) → (∀ {C} → Γ ⊢ C → Δ ⊢ C)
subst σ (` k)          =  σ k
subst σ (ƛ N)          =  ƛ (subst (exts σ) N)
subst σ (L · M)        =  (subst σ L) · (subst σ M)
subst σ (`zero)        =  `zero
subst σ (`suc M)       =  `suc (subst σ M)
subst σ (case L M N)   =  case (subst σ L) (subst σ M) (subst (exts σ) N)
subst σ (μ N)          =  μ (subst (exts σ) N)
subst σ (con n)        =  con n
subst σ (M `* N)       =  subst σ M `* subst σ N
subst σ (`let M N)     =  `let (subst σ M) (subst (exts σ) N)
subst σ `⟨ M , N ⟩     =  `⟨ subst σ M , subst σ N ⟩
subst σ (`proj₁ L)     =  `proj₁ (subst σ L)
subst σ (`proj₂ L)     =  `proj₂ (subst σ L)
subst σ (case× L M)    =  case× (subst σ L) (subst (exts (exts σ)) M)
```

Single and double substitution

```
substZero : ∀ {Γ}{A B} → Γ ⊢ A → Γ , A ∋ B → Γ ⊢ B
substZero V Z      =  V
substZero V (S x)  =  ` x

_[_] : ∀ {Γ A B}
  → Γ , A ⊢ B
  → Γ ⊢ A
    ---------
  → Γ ⊢ B
_[_] {Γ} {A} N V =  subst {Γ , A} {Γ} (substZero V) N

_[_][_] : ∀ {Γ A B C}
  → Γ , A , B ⊢ C
  → Γ ⊢ A
  → Γ ⊢ B
    -------------
  → Γ ⊢ C
_[_][_] {Γ} {A} {B} N V W =  subst {Γ , A , B} {Γ} σ N
  where
  σ : ∀ {C} → Γ , A , B ∋ C → Γ ⊢ C
  σ Z          =  W
  σ (S Z)     = V
  σ (S (S x)) = ` x
```

Values

```
data Value : ∀ {Γ A} → Γ ⊢ A → Set where

  -- functions

  V-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B}
      ---------------------------
    → Value (ƛ N)

  -- naturals

  V-zero : ∀ {Γ}
      -----------------
    → Value (`zero {Γ})

  V-suc_ : ∀ {Γ} {V : Γ ⊢ `ℕ}
    → Value V
      --------------
    → Value (`suc V)

  -- primitives

  V-con : ∀ {Γ n}
      -----------------
    → Value (con {Γ} n)

  -- products

  V-⟨_,_⟩ : ∀ {Γ A B} {V : Γ ⊢ A} {W : Γ ⊢ B}
    → Value V
    → Value W
      ----------------
    → Value `⟨ V , W ⟩
```

Implicit arguments need to be supplied when they are not fixed by the given arguments.

Reduction

```
infix 2 _—→_

data _—→_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  -- functions

  ξ-·₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A ⇒ B} {M : Γ ⊢ A}
    → L —→ L′
      ---------------
    → L · M —→ L′ · M

  ξ-·₂ : ∀ {Γ A B} {V : Γ ⊢ A ⇒ B} {M M′ : Γ ⊢ A}
```

```
    → Value V
    → M —→ M′
      ---------------
    → V · M —→ V · M′

  β-ƛ : ∀ {Γ A B} {N : Γ , A ⊢ B} {V : Γ ⊢ A}
    → Value V
      --------------------
    → (ƛ N) · V —→ N [ V ]

  -- naturals

  ξ-suc : ∀ {Γ} {M M′ : Γ ⊢ `ℕ}
    → M —→ M′
      -----------------
    → `suc M —→ `suc M′

  ξ-case : ∀ {Γ A} {L L′ : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
    → L —→ L′
      -------------------------
    → case L M N —→ case L′ M N

  β-zero : ∀ {Γ A} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
      -------------------
    → case `zero M N —→ M

  β-suc : ∀ {Γ A} {V : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : Γ , `ℕ ⊢ A}
    → Value V
      ----------------------------
    → case (`suc V) M N —→ N [ V ]

  -- fixpoint

  β-μ : ∀ {Γ A} {N : Γ , A ⊢ A}
      ----------------
    → μ N —→ N [ μ N ]

  -- primitive numbers

  ξ-*₁ : ∀ {Γ} {L L′ M : Γ ⊢ Nat}
    → L —→ L′
      -----------------
    → L `* M —→ L′ `* M

  ξ-*₂ : ∀ {Γ} {V M M′ : Γ ⊢ Nat}
    → Value V
    → M —→ M′
      -----------------
    → V `* M —→ V `* M′

  δ-* : ∀ {Γ c d}
      ---------------------------------
    → con {Γ} c `* con d —→ con (c * d)

  -- let

  ξ-let : ∀ {Γ A B} {M M′ : Γ ⊢ A} {N : Γ , A ⊢ B}
    → M —→ M′
      ---------------------
    → `let M N —→ `let M′ N

  β-let : ∀ {Γ A B} {V : Γ ⊢ A} {N : Γ , A ⊢ B}
    → Value V
```

```
      --------------------
    → `let V N —→ N [ V ]

  -- products

  ξ-⟨,⟩₁ : ∀ {Γ A B} {M M′ : Γ ⊢ A} {N : Γ ⊢ B}
    → M —→ M′
      -------------------------
    → `⟨ M , N ⟩ —→ `⟨ M′ , N ⟩

  ξ-⟨,⟩₂ : ∀ {Γ A B} {V : Γ ⊢ A} {N N′ : Γ ⊢ B}
    → Value V
    → N —→ N′
      -------------------------
    → `⟨ V , N ⟩ —→ `⟨ V , N′ ⟩

  ξ-proj₁ : ∀ {Γ A B} {L L′ : Γ ⊢ A `× B}
    → L —→ L′
      ---------------------
    → `proj₁ L —→ `proj₁ L′

  ξ-proj₂ : ∀ {Γ A B} {L L′ : Γ ⊢ A `× B}
    → L —→ L′
      ---------------------
    → `proj₂ L —→ `proj₂ L′

  β-proj₁ : ∀ {Γ A B} {V : Γ ⊢ A} {W : Γ ⊢ B}
    → Value V
    → Value W
      ----------------------
    → `proj₁ `⟨ V , W ⟩ —→ V

  β-proj₂ : ∀ {Γ A B} {V : Γ ⊢ A} {W : Γ ⊢ B}
    → Value V
    → Value W
      ----------------------
    → `proj₂ `⟨ V , W ⟩ —→ W

  -- alternative formulation of products

  ξ-case× : ∀ {Γ A B C} {L L′ : Γ ⊢ A `× B} {M : Γ , A , B ⊢ C}
    → L —→ L′
      -----------------------
    → case× L M —→ case× L′ M

  β-case× : ∀ {Γ A B C} {V : Γ ⊢ A} {W : Γ ⊢ B} {M : Γ , A , B ⊢ C}
    → Value V
    → Value W
      ----------------------------------
    → case× `⟨ V , W ⟩ M —→ M [ V ][ W ]
```

Reflexive and transitive closure

```
infix 2 _—↠_
infix 1 begin_
infixr 2 _—→⟨_⟩_
infix 3 _∎

data _—↠_ {Γ A} : (Γ ⊢ A) → (Γ ⊢ A) → Set where

  _∎ : (M : Γ ⊢ A)
       ------
    → M —↠ M

  _—→⟨_⟩_ : (L : Γ ⊢ A) {M N : Γ ⊢ A}
    → L —→ M
    → M —↠ N
       ------
    → L —↠ N

begin_ : ∀ {Γ A} {M N : Γ ⊢ A}
  → M —↠ N
    ------
  → M —↠ N
begin M—↠N = M—↠N
```

Values do not reduce

```
V¬—→ : ∀ {Γ A} {M N : Γ ⊢ A}
  → Value M
    ----------
  → ¬ (M —→ N)
V¬—→ V-ƛ        ()
V¬—→ V-zero     ()
V¬—→ (V-suc VM) (ξ-suc M—→M′)      = V¬—→ VM M—→M′
V¬—→ V-con      ()
V¬—→ V-⟨ VM , _ ⟩ (ξ-⟨,⟩₁ M—→M′)   = V¬—→ VM M—→M′
V¬—→ V-⟨ _ , VN ⟩ (ξ-⟨,⟩₂ _ N—→N′) = V¬—→ VN N—→N′
```

Progress

```
data Progress {A} (M : ∅ ⊢ A) : Set where

  step : ∀ {N : ∅ ⊢ A}
    → M —→ N
      ----------
    → Progress M

  done :
      Value M
      ----------
    → Progress M
```

```
progress : ∀ {A}
  → (M : ∅ ⊢ A)
    -----------
  → Progress M
progress (` ())
progress (ƛ N)          = done V-ƛ
progress (L · M) with progress L
... | step L—→L′        = step (ξ-·₁ L—→L′)
... | done V-ƛ with progress M
... | step M—→M′        = step (ξ-·₂ V-ƛ M—→M′)
... | done VM           = step (β-ƛ VM)
progress (`zero)        = done V-zero
progress (`suc M) with progress M
... | step M—→M′        = step (ξ-suc M—→M′)
... | done VM           = done (V-suc VM)
progress (case L M N) with progress L
... | step L—→L′        = step (ξ-case L—→L′)
... | done V-zero       = step β-zero
... | done (V-suc VL)   = step (β-suc VL)
progress (μ N)          = step β-μ
progress (con n)        = done V-con
progress (L `* M) with progress L
... | step L—→L′        = step (ξ-*₁ L—→L′)
... | done V-con with progress M
... | step M—→M′        = step (ξ-*₂ V-con M—→M′)
... | done V-con        = step δ-*
progress (`let M N) with progress M
... | step M—→M′        = step (ξ-let M—→M′)
... | done VM           = step (β-let VM)
progress `⟨ M , N ⟩ with progress M
... | step M—→M′        = step (ξ-⟨,⟩₁ M—→M′)
... | done VM with progress N
... | step N—→N′        = step (ξ-⟨,⟩₂ VM N—→N′)
... | done VN           = done (V-⟨ VM , VN ⟩)
progress (`proj₁ L) with progress L
... | step L—→L′        = step (ξ-proj₁ L—→L′)
... | done (V-⟨ VM , VN ⟩) = step (β-proj₁ VM VN)
progress (`proj₂ L) with progress L
... | step L—→L′        = step (ξ-proj₂ L—→L′)
... | done (V-⟨ VM , VN ⟩) = step (β-proj₂ VM VN)
progress (case× L M) with progress L
... | step L—→L′        = step (ξ-case× L—→L′)
... | done (V-⟨ VM , VN ⟩) = step (β-case× VM VN)
```

Evaluation

```
record Gas : Set where
  constructor gas
  field
    amount : ℕ

data Finished {Γ A} (N : Γ ⊢ A) : Set where

  done :
      Value N
      ----------
    → Finished N

  out-of-gas :
     ----------
     Finished N

data Steps {A} : ∅ ⊢ A → Set where

  steps : {L N : ∅ ⊢ A}
    → L —↠ N
    → Finished N
      ----------
    → Steps L

eval : ∀ {A}
  → Gas
  → (L : ∅ ⊢ A)
    -----------
  → Steps L
eval (gas zero) L    = steps (L ∎) out-of-gas
eval (gas (suc m)) L with progress L
... | done VL        = steps (L ∎) (done VL)
... | step {M} L—→M with eval (gas m) M
... | steps M—↠N fin = steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

Examples

```
cube : ∅ ⊢ Nat ⇒ Nat
cube = ƛ (# 0 `* # 0 `* # 0)

_ : cube · con 2 —↠ con 8
_ =
  begin
    cube · con 2
  —→⟨ β-ƛ V-con ⟩
    con 2 `* con 2 `* con 2
  —→⟨ ξ-*₁ δ-* ⟩
    con 4 `* con 2
  —→⟨ δ-* ⟩
    con 8
  ∎

exp10 : ∅ ⊢ Nat ⇒ Nat
exp10 = ƛ (`let (# 0 `* # 0)
            (`let (# 0 `* # 0)
              (`let (# 0 `* # 2)
                (# 0 `* # 0))))

_ : exp10 · con 2 —↠ con 1024
_ =
  begin
    exp10 · con 2
  —→⟨ β-ƛ V-con ⟩
    `let (con 2 `* con 2) (`let (# 0 `* # 0) (`let (# 0 `* con 2) (# 0 `* # 0)))
  —→⟨ ξ-let δ-* ⟩
    `let (con 4) (`let (# 0 `* # 0) (`let (# 0 `* con 2) (# 0 `* # 0)))
  —→⟨ β-let V-con ⟩
```

```
    `let (con 4 `* con 4) (`let (# 0 `* con 2) (# 0 `* # 0))
  —→⟨ ξ-let δ-* ⟩
    `let (con 16) (`let (# 0 `* con 2) (# 0 `* # 0))
  —→⟨ β-let V-con ⟩
    `let (con 16 `* con 2) (# 0 `* # 0)
  —→⟨ ξ-let δ-* ⟩
    `let (con 32) (# 0 `* # 0)
  —→⟨ β-let V-con ⟩
    con 32 `* con 32
  —→⟨ δ-* ⟩
    con 1024
  ∎

swap× : ∀ {A B} → ∅ ⊢ A `× B ⇒ B `× A
swap× = ƛ `⟨ `proj₂ (# 0) , `proj₁ (# 0) ⟩

_ : swap× · `⟨ con 42 , `zero ⟩ —↠ `⟨ `zero , con 42 ⟩
_ =
  begin
    swap× · `⟨ con 42 , `zero ⟩
  —→⟨ β-ƛ V-⟨ V-con , V-zero ⟩ ⟩
    `⟨ `proj₂ `⟨ con 42 , `zero ⟩ , `proj₁ `⟨ con 42 , `zero ⟩ ⟩
  —→⟨ ξ-⟨,⟩₁ (β-proj₂ V-con V-zero) ⟩
    `⟨ `zero , `proj₁ `⟨ con 42 , `zero ⟩ ⟩
  —→⟨ ξ-⟨,⟩₂ V-zero (β-proj₁ V-con V-zero) ⟩
    `⟨ `zero , con 42 ⟩
  ∎

swap×-case : ∀ {A B} → ∅ ⊢ A `× B ⇒ B `× A
swap×-case = ƛ case× (# 0) `⟨ # 0 , # 1 ⟩

_ : swap×-case · `⟨ con 42 , `zero ⟩ —↠ `⟨ `zero , con 42 ⟩
_ =
  begin
     swap×-case · `⟨ con 42 , `zero ⟩
   —→⟨ β-ƛ V-⟨ V-con , V-zero ⟩ ⟩
     case× `⟨ con 42 , `zero ⟩ `⟨ # 0 , # 1 ⟩
   —→⟨ β-case× V-con V-zero ⟩
     `⟨ `zero , con 42 ⟩
   ∎
```

Exercise `More` (recommended and practice)

Formalise the remaining constructs defined in this chapter. Make your changes in this file. Evaluate each example, applied to data as needed, to confirm it returns the expected answer:

- sums (recommended)
- unit type (practice)
- an alternative formulation of unit type (practice)
- empty type (recommended)
- lists (practice)

Please delimit any code you add as follows:

```
-- begin
-- end
```

Exercise `double-subst` **(stretch)**

Show that a double substitution is equivalent to two single substitutions.

```
postulate
  double-subst :
    ∀ {Γ A B C} {V : Γ ⊢ A} {W : Γ ⊢ B} {N : Γ , A , B ⊢ C} →
      N [ V ][ W ] ≡ (N [ rename S_ W ]) [ V ]
```

Note the arguments need to be swapped and `W` needs to have its context adjusted via renaming in order for the right-hand side to be well typed.

Test examples

We repeat the test examples from Chapter DeBruijn, in order to make sure we have not broken anything in the process of extending our base calculus.

```
two : ∀ {Γ} → Γ ⊢ `ℕ
two = `suc `suc `zero

plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = μ ƛ ƛ (case (# 1) (# 0) (`suc (# 3 · # 0 · # 1)))

2+2 : ∀ {Γ} → Γ ⊢ `ℕ
2+2 = plus · two · two

Ch : Type → Type
Ch A = (A ⇒ A) ⇒ A ⇒ A

twoᶜ : ∀ {Γ A} → Γ ⊢ Ch A
twoᶜ = ƛ ƛ (# 1 · (# 1 · # 0))

plusᶜ : ∀ {Γ A} → Γ ⊢ Ch A ⇒ Ch A ⇒ Ch A
plusᶜ = ƛ ƛ ƛ ƛ (# 3 · # 1 · (# 2 · # 1 · # 0))

sucᶜ : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ
sucᶜ = ƛ `suc (# 0)

2+2ᶜ : ∀ {Γ} → Γ ⊢ `ℕ
2+2ᶜ = plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

Unicode

This chapter uses the following unicode:

```
σ  U+03C3  GREEK SMALL LETTER SIGMA (\Gs or \sigma)
†  U+2020  DAGGER (\dag)
‡  U+2021  DOUBLE DAGGER (\ddag)
```

Chapter 15

Bisimulation: Relating reduction systems

```
module plfa.part2.Bisimulation where
```

Some constructs can be defined in terms of other constructs. In the previous chapter, we saw how *let* terms can be rewritten as an application of an abstraction, and how two alternative formulations of products — one with projections and one with case — can be formulated in terms of each other. In this chapter, we look at how to formalise such claims.

Given two different systems, with different terms and reduction rules, we define what it means to claim that one *simulates* the other. Let's call our two systems *source* and *target*. Let `M`, `N` range over terms of the source, and `M†`, `N†` range over terms of the target. We define a relation

```
M ~ M†
```

between corresponding terms of the two systems. We have a *simulation* of the source by the target if every reduction in the source has a corresponding reduction sequence in the target:

Simulation: For every `M`, `M†`, and `N`: If `M ~ M†` and `M —→ N` then `M† —↠ N†` and `N ~ N†` for some `N†`.

Or, in a diagram:

```
M  --- —→ --- N
|             |
|             |
~             ~
|             |
|             |
M† --- —↠ --- N†
```

Sometimes we will have a stronger condition, where each reduction in the source corresponds to a reduction (rather than a reduction sequence) in the target:

```
M  --- —→ --- N
|             |
|             |
~             ~
|             |
|             |
M† --- ⟶ --- N†
```

This stronger condition is known as *lock-step* or *on the nose* simulation.

We are particularly interested in the situation where there is also a simulation from the target to the source: every reduction in the target has a corresponding reduction sequence in the source. This situation is called a *bisimulation*.

Simulation is established by case analysis over all possible reductions and all possible terms to which they are related. For each reduction step in the source we must show a corresponding reduction sequence in the target.

For instance, the source might be lambda calculus with *let* added, and the target the same system with `let` translated out. The key rule defining our relation will be:

```
M ~ M†
N ~ N†
--------------------------------
let x = M in N ~ (ƛ x ⇒ N†) · M†
```

All the other rules are congruences: variables relate to themselves, and abstractions and applications relate if their components relate:

```
-----
x ~ x

N ~ N†
------------------
ƛ x ⇒ N ~ ƛ x ⇒ N†

L ~ L†
M ~ M†
---------------
L · M ~ L† · M†
```

Covering the other constructs of our language — naturals, fixpoints, products, and so on — would add little save length.

In this case, our relation can be specified by a function from source to target:

```
(x) †                =  x
(ƛ x ⇒ N) †          =  ƛ x ⇒ (N †)
(L · M) †            =  (L †) · (M †)
(let x = M in N) †   =  (ƛ x ⇒ (N †)) · (M †)
```

And we have

```
M † ≡ N
-------
M ~ N
```

and conversely. But in general we may have a relation without any corresponding function.

This chapter formalises establishing that `~` as defined above is a simulation from source to target. We leave establishing it in the reverse direction as an exercise. Another exercise is to show the alternative formulations of products in Chapter More are in bisimulation.

Imports

We import our source language from Chapter More:

```
open import plfa.part2.More
```

Simulation

The simulation is a straightforward formalisation of the rules in the introduction:

```
infix 4 _~_
infix 5 ~ƛ_
infix 7 _~·_

data _~_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  ~` : ∀ {Γ A} {x : Γ ∋ A}
     ---------
   → ` x ~ ` x

  ~ƛ_ : ∀ {Γ A B} {N N† : Γ , A ⊢ B}
    → N ~ N†
      ----------
    → ƛ N ~ ƛ N†

  _~·_ : ∀ {Γ A B} {L L† : Γ ⊢ A ⇒ B} {M M† : Γ ⊢ A}
    → L ~ L†
    → M ~ M†
      ---------------
    → L · M ~ L† · M†

  ~let : ∀ {Γ A B} {M M† : Γ ⊢ A} {N N† : Γ , A ⊢ B}
    → M ~ M†
    → N ~ N†
      ----------------------
    → `let M N ~ (ƛ N†) · M†
```

The language in Chapter More has more constructs, which we could easily add. However, leaving the simulation small lets us focus on the essence. It's a handy technical trick that we can have a large source language, but only bother to include in the simulation the terms of interest.

Exercise `_†` (practice)

Formalise the translation from source to target given in the introduction. Show that `M † ≡ N` implies `M ~ N`, and conversely.

Hint: For simplicity, we focus on only a few constructs of the language, so `_†` should be defined only on relevant terms. One way to do this is to use a decidable predicate to pick out terms in the domain of `_†`, using proof by reflection.

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

Simulation commutes with values

We need a number of technical results. The first is that simulation commutes with values. That is, if `M ~ M†` and `M` is a value then `M†` is also a value:

```
~val : ∀ {Γ A} {M M† : Γ ⊢ A}
  → M ~ M†
  → Value M
    --------
  → Value M†
~val ~`           ()
~val (~ƛ ~N)      V-ƛ = V-ƛ
~val (~L ~· ~M) ()
~val (~let ~M ~N) ()
```

It is a straightforward case analysis, where here the only value of interest is a lambda abstraction.

Exercise `~val⁻¹` (practice)

Show that this also holds in the reverse direction: if `M ~ M†` and `Value M†` then `Value M`.

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

Simulation commutes with renaming

The next technical result is that simulation commutes with renaming. That is, if `ρ` maps any judgment `Γ ∋ A` to a judgment `Δ ∋ A`, and if `M ~ M†` then `rename ρ M ~ rename ρ M†`:

```
~rename : ∀ {Γ Δ}
  → (ρ : ∀ {A} → Γ ∋ A → Δ ∋ A)
    ----------------------------------------------------------
  → (∀ {A} {M M† : Γ ⊢ A} → M ~ M† → rename ρ M ~ rename ρ M†)
~rename ρ (~`)         = ~`
~rename ρ (~ƛ ~N)      = ~ƛ (~rename (ext ρ) ~N)
~rename ρ (~L ~· ~M)   = (~rename ρ ~L) ~· (~rename ρ ~M)
~rename ρ (~let ~M ~N) = ~let (~rename ρ ~M) (~rename (ext ρ) ~N)
```

The structure of the proof is similar to the structure of renaming itself: reconstruct each term with recursive invocation, extending the environment where appropriate (in this case, only for the body of an abstraction).

Simulation commutes with substitution

The third technical result is that simulation commutes with substitution. It is more complex than renaming, because where we had one renaming map `ρ` here we need two substitution maps, `σ` and `σ†`.

The proof first requires we establish an analogue of extension. If `σ` and `σ†` both map any judgment `Γ ∋ A` to a judgment `Δ ⊢ A`, such that for every `x` in `Γ ∋ A` we have `σ x ~ σ† x`, then for any `x` in `Γ , B ∋ A` we have `exts σ x ~ exts σ† x`:

```
~exts : ∀ {Γ Δ}
  → {σ : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → {σ† : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → (∀ {A} → (x : Γ ∋ A) → σ x ~ σ† x)
    --------------------------------------------------
  → (∀ {A B} → (x : Γ , B ∋ A) → exts σ x ~ exts σ† x)
~exts ~σ Z      = ~`
~exts ~σ (S x) = ~rename S_ (~σ x)
```

The structure of the proof is similar to the structure of extension itself. The newly introduced variable trivially relates to itself, and otherwise we apply renaming to the hypothesis.

With extension under our belts, it is straightforward to show substitution commutes. If `σ` and `σ†` both map any judgment `Γ ∋ A` to a judgment `Δ ⊢ A`, such that for every `x` in `Γ ∋ A` we have `σ x ~ σ† x`, and if `M ~ M†`, then `subst σ M ~ subst σ† M†`:

```
~subst : ∀ {Γ Δ}
  → {σ : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → {σ† : ∀ {A} → Γ ∋ A → Δ ⊢ A}
  → (∀ {A} → (x : Γ ∋ A) → σ x ~ σ† x)
    ---------------------------------------------------------
  → (∀ {A} {M M† : Γ ⊢ A} → M ~ M† → subst σ M ~ subst σ† M†)
~subst ~σ (~` {x = x})  = ~σ x
~subst ~σ (~ƛ ~N)       = ~ƛ (~subst (~exts ~σ) ~N)
~subst ~σ (~L ~· ~M)    = (~subst ~σ ~L) ~· (~subst ~σ ~M)
~subst ~σ (~let ~M ~N) = ~let (~subst ~σ ~M) (~subst (~exts ~σ) ~N)
```

Again, the structure of the proof is similar to the structure of substitution itself: reconstruct each term with recursive invocation, extending the environment where appropriate (in this case, only for the body of an abstraction).

From the general case of substitution, it is also easy to derive the required special case. If `N ~ N†` and `M ~ M†`, then `N [ M ] ~ N† [ M† ]`:

```
~sub : ∀ {Γ A B} {N N† : Γ , B ⊢ A} {M M† : Γ ⊢ B}
  → N ~ N†
  → M ~ M†
    ------------------------
  → (N [ M ]) ~ (N† [ M† ])
~sub {Γ} {A} {B} ~N ~M = ~subst {Γ , B} {Γ} ~σ {A} ~N
  where
  ~σ : ∀ {A} → (x : Γ , B ∋ A) → _ ~ _
  ~σ Z      = ~M
  ~σ (S x) = ~`
```

Once more, the structure of the proof resembles the original.

The relation is a simulation

Finally, we can show that the relation actually is a simulation. In fact, we will show the stronger condition of a lock-step simulation. What we wish to show is:

Lock-step simulation: For every `M`, `M†`, and `N`: If `M ~ M†` and `M —→ N` then `M† —→ N†` and `N ~ N†` for some `N†`.

Or, in a diagram:

```
M  --- ⟶ --- N
|            |
|            |
~            ~
|            |
|            |
M† --- ⟶ --- N†
```

We first formulate a concept corresponding to the lower leg of the diagram, that is, its right and bottom edges:

```
data Leg {Γ A} (M† N : Γ ⊢ A) : Set where

  leg : ∀ {N† : Γ ⊢ A}
    → N ~ N†
    → M† ⟶ N†
      --------
    → Leg M† N
```

For our formalisation, in this case, we can use a stronger relation than `—↠`, replacing it by `⟶`.

We can now state and prove that the relation is a simulation. Again, in this case, we can use a stronger relation than `—↠`, replacing it by `⟶`:

```
sim : ∀ {Γ A} {M M† N : Γ ⊢ A}
  → M ~ M†
  → M ⟶ N
    ---------
  → Leg M† N
sim ~`           ()
sim (~ƛ ~N)      ()
sim (~L ~· ~M)   (ξ-·₁ L⟶)
  with sim ~L L⟶
... | leg ~L′ L†⟶           = leg (~L′ ~· ~M)  (ξ-·₁ L†⟶)
sim (~V ~· ~M)   (ξ-·₂ VV M⟶)
  with sim ~M M⟶
... | leg ~M′ M†⟶           = leg (~V ~· ~M′)  (ξ-·₂ (~val ~V VV) M†⟶)
sim ((~ƛ ~N) ~· ~V) (β-ƛ VV) = leg (~sub ~N ~V) (β-ƛ (~val ~V VV))
sim (~let ~M ~N) (ξ-let M⟶)
  with sim ~M M⟶
... | leg ~M′ M†⟶           = leg (~let ~M′ ~N) (ξ-·₂ V-ƛ M†⟶)
sim (~let ~V ~N) (β-let VV)  = leg (~sub ~N ~V) (β-ƛ (~val ~V VV))
```

The proof is by case analysis, examining each possible instance of `M ~ M†` and each possible instance of `M ⟶ M†`, using recursive invocation whenever the reduction is by a `ξ` rule, and hence contains another reduction. In its structure, it looks a little bit like a proof of progress:

- If the related terms are variables, no reduction applies.
- If the related terms are abstractions, no reduction applies.
- If the related terms are applications, there are three subcases:
  - The source term reduces via `ξ-·₁`, in which case the target term does as well. Recursive invocation gives us

```
  L  --- —→ ---  L′
  |              |
  |              |
  ~              ~
  |              |
  |              |
  L† --- —→ --- L′†
```

from which follows:

```
   L · M  --- —→ ---  L′ · M
     |                  |
     |                  |
     ~                  ~
     |                  |
     |                  |
  L† · M† --- —→ --- L′† · M†
```

- The source term reduces via `ξ-·₂`, in which case the target term does as well. Recursive invocation gives us

```
  M  --- —→ ---  M′
  |              |
  |              |
  ~              ~
  |              |
  |              |
  M† --- —→ --- M′†
```

from which follows:

```
   V · M  --- —→ ---  V · M′
     |                  |
     |                  |
     ~                  ~
     |                  |
     |                  |
  V† · M† --- —→ --- V† · M′†
```

Since simulation commutes with values and `V` is a value, `V†` is also a value.

- The source term reduces via `β-ƛ`, in which case the target term does as well:

```
   (ƛ x ⇒ N) · V  --- —→ ---  N [ x := V ]
        |                           |
        |                           |
        ~                           ~
        |                           |
        |                           |
  (ƛ x ⇒ N†) · V† --- —→ --- N† [ x :=  V† ]
```

Since simulation commutes with values and `V` is a value, `V†` is also a value. Since simulation commutes with substitution and `N ~ N†` and `V ~ V†`, we have `N [ x := V ] ~ N† [ x := V† ]`.

• If the related terms are a let and an application of an abstraction, there are two subcases:

- The source term reduces via `ξ-let`, in which case the target term reduces via `ξ-·₂`. Recursive invocation gives us

```
    M  --- ⟶ ---  M′
    |             |
    |             |
    ~             ~
    |             |
    |             |
    M† --- ⟶ --- M′†
```

from which follows:

```
    let x = M in N --- ⟶ --- let x = M′ in N
          |                        |
          |                        |
          ~                        ~
          |                        |
          |                        |
    (ƛ x ⇒ N) · M  --- ⟶ --- (ƛ x ⇒ N) · M′
```

- The source term reduces via `β-let`, in which case the target term reduces via `β-ƛ`:

```
    let x = V in N  --- ⟶ ---  N [ x := V ]
          |                        |
          |                        |
          ~                        ~
          |                        |
          |                        |
    (ƛ x ⇒ N†) · V† --- ⟶ --- N† [ x := V† ]
```

  Since simulation commutes with values and `V` is a value, `V†` is also a value. Since simulation commutes with substitution and `N ~ N†` and `V ~ V†`, we have `N [ x := V ] ~ N† [ x := V† ]`.

Exercise `sim⁻¹` (practice)

Show that we also have a simulation in the other direction, and hence that we have a bisimulation.

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

Exercise `products` (practice)

Show that the two formulations of products in Chapter More are in bisimulation. The only constructs you need to include are variables, and those connected to functions and products. In this case, the simulation is *not* lock-step.

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

Unicode

This chapter uses the following unicode:

```
†  U+2020  DAGGER (\dag)
⁻  U+207B  SUPERSCRIPT MINUS (\^-)
¹  U+00B9  SUPERSCRIPT ONE (\^1)
```

Chapter 16

Inference: Bidirectional type inference

```
module plfa.part2.Inference where
```

So far in our development, type derivations for the corresponding term have been provided by fiat. In Chapter Lambda type derivations are extrinsic to the term, while in Chapter DeBruijn type derivations are intrinsic to the term, but in both we have written out the type derivations in full.

In practice, one often writes down a term with a few decorations and applies an algorithm to *infer* the corresponding type derivation. Indeed, this is exactly what happens in Agda: we specify the types for top-level function declarations, and type information for everything else is inferred from what has been given. The style of inference Agda uses is based on a technique called *bidirectional* type inference, which will be presented in this chapter.

This chapter ties our previous developments together. We begin with a term with some type annotations, close to the raw terms of Chapter Lambda, and from it we compute an intrinsically-typed term, in the style of Chapter DeBruijn.

Introduction: Inference rules as algorithms

In the calculus we have considered so far, a term may have more than one type. For example,

```
(ƛ x ⇒ x) ⦂ (A ⇒ A)
```

holds for *every* type `A`. We start by considering a small language for lambda terms where every term has a unique type. All we need do is decorate each abstraction term with the type of its argument. This gives us the grammar:

```
L, M, N ::=                         decorated terms
  x                                   variable
  ƛ x ⦂ A ⇒ N                         abstraction (decorated)
  L · M                               application
```

Each of the associated type rules can be read as an algorithm for type checking. For each typing judgment, we label each position as either an *input* or an *output*.

For the judgment

```
Γ ∋ x ⦂ A
```

we take the context `Γ` and the variable `x` as inputs, and the type `A` as output. Consider the rules:

```
----------------- Z
Γ , x ⦂ A ∋ x ⦂ A

Γ ∋ x ⦂ A
----------------- S
Γ , y ⦂ B ∋ x ⦂ A
```

From the inputs we can determine which rule applies: if the last variable in the context matches the given variable then the first rule applies, else the second. (For de Bruijn indices, it is even easier: zero matches the first rule and successor the second.) For the first rule, the output type can be read off as the last type in the input context. For the second rule, the inputs of the conclusion determine the inputs of the hypothesis, and the output of the hypothesis determines the output of the conclusion.

For the judgment

```
Γ ⊢ M ⦂ A
```

we take the context `Γ` and term `M` as inputs, and the type `A` as output. Consider the rules:

```
Γ ∋ x ⦂ A
-----------
Γ ⊢ ` x ⦂ A

Γ , x ⦂ A ⊢ N ⦂ B
---------------------------
Γ ⊢ (ƛ x ⦂ A ⇒ N) ⦂ (A ⇒ B)

Γ ⊢ L ⦂ A ⇒ B
Γ ⊢ M ⦂ A′
A ≡ A′
-------------
Γ ⊢ L · M ⦂ B
```

The input term determines which rule applies: variables use the first rule, abstractions the second, and applications the third. We say such rules are *syntax directed*. For the variable rule, the inputs of the conclusion determine the inputs of the hypothesis, and the output of the hypothesis determines the output of the conclusion. Same for the abstraction rule — the bound variable and argument are carried from the term of the conclusion into the context of the hypothesis; this works because we added the argument type to the abstraction. For the application rule, we add a third hypothesis to check whether the domain of the function matches the type of the argument; this judgment is decidable when both types are given as inputs. The inputs of the conclusion determine the inputs of the first two hypotheses, the outputs of the first two hypotheses determine the inputs of the third hypothesis, and the output of the first hypothesis determines the output of the conclusion.

Converting the above to an algorithm is straightforward, as is adding naturals and fixpoint. We omit the details. Instead, we consider a detailed description of an approach that requires less obtrusive decoration. The idea is to break the normal typing judgment into two judgments, one that produces the type as an output (as above), and another that takes it as an input.

Synthesising and inheriting types

In addition to the lookup judgment for variables, which will remain as before, we now have two judgments for the type of the term:

```
Γ ⊢ M ↑ A
Γ ⊢ M ↓ A
```

The first of these *synthesises* the type of a term, as before, while the second *inherits* the type. In the first, the context and term are inputs and the type is an output; while in the second, all three of the context, term, and type are inputs.

Which terms use synthesis and which inheritance? Our approach will be that the main term in a *deconstructor* is typed via synthesis while *constructors* are typed via inheritance. For instance, the function in an application is typed via synthesis, but an abstraction is typed via inheritance. The inherited type in an abstraction term serves the same purpose as the argument type decoration of the previous section.

Terms that deconstruct a value of a type always have a main term (supplying an argument of the required type) and often have side-terms. For application, the main term supplies the function and the side term supplies the argument. For case terms, the main term supplies a natural and the side terms are the two branches. In a deconstructor, the main term will be typed using synthesis but the side terms will be typed using inheritance. As we will see, this leads naturally to an application as a whole being typed by synthesis, while a case term as a whole will be typed by inheritance. Variables are naturally typed by synthesis, since we can look up the type in the input context. Fixed points will be naturally typed by inheritance.

In order to get a syntax-directed type system we break terms into two kinds, `Term⁺` and `Term⁻`, which are typed by synthesis and inheritance, respectively. A subterm that is typed by synthesis may appear in a context where it is typed by inheritance, or vice-versa, and this gives rise to two new term forms.

For instance, we said above that the argument of an application is typed by inheritance and that variables are typed by synthesis, giving a mismatch if the argument of an application is a variable. Hence, we need a way to treat a synthesized term as if it is inherited. We introduce a new term form, `M ↑` for this purpose. The typing judgment checks that the inherited and synthesised types match.

Similarly, we said above that the function of an application is typed by synthesis and that abstractions are typed by inheritance, giving a mismatch if the function of an application is an abstraction. Hence, we need a way to treat an inherited term as if it is synthesised. We introduce a new term form `M ↓ A` for this purpose. The typing judgment returns `A` as the synthesized type of the term as a whole, as well as using it as the inherited type for `M`.

The term form `M ↓ A` represents the only place terms need to be decorated with types. It only appears when switching from synthesis to inheritance, that is, when a term that *deconstructs* a value of a type contains as its main term a term that *constructs* a value of a type, in other words, a place where a `β`-reduction will occur. Typically, we will find that decorations are only required on top level declarations.

We can extract the grammar for terms from the above:

```
L⁺, M⁺, N⁺ ::=                    terms with synthesized type
  x                                 variable
  L⁺ · M⁻                           application
  M⁻ ↓ A                            switch to inherited

L⁻, M⁻, N⁻ ::=                    terms with inherited type
  ƛ x ⇒ N⁻                          abstraction
  `zero                             zero
  `suc M⁻                           successor
  case L⁺ [zero⇒ M⁻ |suc x ⇒ N⁻ ]   case
  μ x ⇒ N⁻                          fixpoint
  M⁺ ↑                              switch to synthesized
```

We will formalise the above shortly.

Soundness and completeness

What we intend to show is that the typing judgments are *decidable*:

```
synthesize : ∀ (Γ : Context) (M : Term⁺)
    -----------------------
  → Dec (∃[ A ]( Γ ⊢ M ↑ A ))

inherit : ∀ (Γ : Context) (M : Term⁻) (A : Type)
          ---------------
        → Dec (Γ ⊢ M ↓ A)
```

Given context `Γ` and synthesised term `M`, we must decide whether there exists a type `A` such that `Γ ⊢ M ↑ A` holds, or its negation. Similarly, given context `Γ`, inherited term `M`, and type `A`, we must decide whether `Γ ⊢ M ↓ A` holds, or its negation.

Our proof is constructive. In the synthesised case, it will either deliver a pair of a type `A` and evidence that `Γ ⊢ M ↓ A`, or a function that given such a pair produces evidence of a contradiction. In the inherited case, it will either deliver evidence that `Γ ⊢ M ↑ A`, or a function that given such evidence produces evidence of a contradiction. The positive case is referred to as *soundness* — synthesis and inheritance succeed only if the corresponding relation holds. The negative case is referred to as *completeness* — synthesis and inheritance fail only when they cannot possibly succeed.

Another approach might be to return a derivation if synthesis or inheritance succeeds, and an error message otherwise — for instance, see the section of the Agda user manual discussing syntactic sugar. Such an approach demonstrates soundness, but not completeness. If it returns a derivation, we know it is correct; but there is nothing to prevent us from writing a function that *always* returns an error, even when there exists a correct derivation. Demonstrating both soundness and completeness is significantly stronger than demonstrating soundness alone. The negative proof can be thought of as a semantically verified error message, although in practice it may be less readable than a well-crafted error message.

We are now ready to begin the formal development.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; trans; cong; cong₂; _≢_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc; _+_; _*_)
open import Data.String using (String; _≟_)
open import Data.Product using (_×_; ∃; ∃-syntax) renaming (_,_ to ⟨_,_⟩)
open import Relation.Nullary using (¬_; Dec; yes; no)
```

Once we have a type derivation, it will be easy to construct from it the intrinsically-typed representation. In order that we can compare with our previous development, we import module `plfa.part2.More`:

```
import plfa.part2.More as DB
```

The phrase `as DB` allows us to refer to definitions from that module as, for instance, `DB._⊢_`, which is invoked as `Γ DB.⊢ A`, where `Γ` has type `DB.Context` and `A` has type `DB.Type`.

Syntax

First, we get all our infix declarations out of the way. We list separately operators for judgments and terms:

```
infix  4 _∋_⦂_
infix  4 _⊢_↑_
infix  4 _⊢_↓_
infixl 5 _,_⦂_

infixr 7 _⇒_

infix  5 ƛ_⇒_
infix  5 μ_⇒_
infix  6 _↑
infix  6 _↓_
infixl 7 _·_
infix  8 `suc_
infix  9 `_
```

Identifiers, types, and contexts are as before:

```
Id : Set
Id = String

data Type : Set where
  `ℕ  : Type
  _⇒_ : Type → Type → Type

data Context : Set where
  ∅     : Context
  _,_⦂_ : Context → Id → Type → Context
```

The syntax of terms is defined by mutual recursion. We use `Term⁺` and `Term⁻` for terms with synthesized and inherited types, respectively. Note the inclusion of the switching forms, `M ↓ A` and `M ↑`:

```
data Term⁺ : Set
data Term⁻ : Set

data Term⁺ where
  `_  : Id → Term⁺
  _·_ : Term⁺ → Term⁻ → Term⁺
  _↓_ : Term⁻ → Type → Term⁺

data Term⁻ where
  ƛ_⇒_                    : Id → Term⁻ → Term⁻
  `zero                  : Term⁻
  `suc_                  : Term⁻ → Term⁻
  `case_[zero⇒_|suc_⇒_] : Term⁺ → Term⁻ → Id → Term⁻ → Term⁻
  μ_⇒_                   : Id → Term⁻ → Term⁻
  _↑                     : Term⁺ → Term⁻
```

The choice as to whether each term is synthesized or inherited follows the discussion above, and can be read off from the informal grammar presented earlier. Main terms in deconstructors synthesise, constructors and side terms in deconstructors inherit.

Example terms

We can recreate the examples from preceding chapters. First, computing two plus two on naturals:

```
two : Term⁻
two = `suc (`suc `zero)

plus : Term⁺
plus = (μ "p" ⇒ ƛ "m" ⇒ ƛ "n" ⇒
          `case (` "m") [zero⇒ ` "n" ↑
                        |suc "m" ⇒ `suc (` "p" · (` "m" ↑) · (` "n" ↑) ↑) ])
            ↓ (`ℕ ⇒ `ℕ ⇒ `ℕ)

2+2 : Term⁺
2+2 = plus · two · two
```

The only change is to decorate with down and up arrows as required. The only type decoration required is for `plus`.

Next, computing two plus two with Church numerals:

```
Ch : Type
Ch = (`ℕ ⇒ `ℕ) ⇒ `ℕ ⇒ `ℕ

twoᶜ : Term⁻
twoᶜ = (ƛ "s" ⇒ ƛ "z" ⇒ ` "s" · (` "s" · (` "z" ↑) ↑) ↑)

plusᶜ : Term⁺
plusᶜ = (ƛ "m" ⇒ ƛ "n" ⇒ ƛ "s" ⇒ ƛ "z" ⇒
           ` "m" · (` "s" ↑) · (` "n" · (` "s" ↑) · (` "z" ↑) ↑) ↑)
             ↓ (Ch ⇒ Ch ⇒ Ch)

sucᶜ : Term⁻
sucᶜ = ƛ "x" ⇒ `suc (` "x" ↑)

2+2ᶜ : Term⁺
2+2ᶜ = plusᶜ · twoᶜ · twoᶜ · sucᶜ · `zero
```

The only type decoration required is for `plusᶜ`. One is not even required for `sucᶜ`, which inherits its type as an argument of `plusᶜ`.

Bidirectional type checking

The typing rules for variables are as in Lambda:

```
data _∋_⦂_ : Context → Id → Type → Set where

  Z : ∀ {Γ x A}
      --------------------
    → Γ , x ⦂ A ∋ x ⦂ A

  S : ∀ {Γ x y A B}
    → x ≢ y
    → Γ ∋ x ⦂ A
      -----------------
    → Γ , y ⦂ B ∋ x ⦂ A
```

As with syntax, the judgments for synthesizing and inheriting types are mutually recursive:

```
data _⊢_↑_ : Context → Term⁺ → Type → Set
data _⊢_↓_ : Context → Term⁻ → Type → Set

data _⊢_↑_ where

  ⊢` : ∀ {Γ A x}
    → Γ ∋ x ⦂ A
      -----------
    → Γ ⊢ ` x ↑ A

  _·_ : ∀ {Γ L M A B}
    → Γ ⊢ L ↑ A ⇒ B
    → Γ ⊢ M ↓ A
      -------------
    → Γ ⊢ L · M ↑ B

  ⊢↓ : ∀ {Γ M A}
    → Γ ⊢ M ↓ A
      ---------------
    → Γ ⊢ (M ↓ A) ↑ A

data _⊢_↓_ where

  ⊢ƛ : ∀ {Γ x N A B}
    → Γ , x ⦂ A ⊢ N ↓ B
      -------------------
    → Γ ⊢ ƛ x ⇒ N ↓ A ⇒ B

  ⊢zero : ∀ {Γ}
      --------------
    → Γ ⊢ `zero ↓ `ℕ

  ⊢suc : ∀ {Γ M}
    → Γ ⊢ M ↓ `ℕ
      ---------------
    → Γ ⊢ `suc M ↓ `ℕ

  ⊢case : ∀ {Γ L M x N A}
    → Γ ⊢ L ↑ `ℕ
    → Γ ⊢ M ↓ A
    → Γ , x ⦂ `ℕ ⊢ N ↓ A
      -------------------------------------
    → Γ ⊢ `case L [zero⇒ M |suc x ⇒ N ] ↓ A

  ⊢μ : ∀ {Γ x N A}
    → Γ , x ⦂ A ⊢ N ↓ A
      -----------------
    → Γ ⊢ μ x ⇒ N ↓ A

  ⊢↑ : ∀ {Γ M A B}
    → Γ ⊢ M ↑ A
    → A ≡ B
      -------------
    → Γ ⊢ (M ↑) ↓ B
```

We follow the same convention as Chapter Lambda, prefacing the constructor with `⊢` to derive the name of the corresponding type rule.

The rules are similar to those in Chapter Lambda, modified to support synthesised and inherited types. The two new rules are those for `⊢↓` and `⊢↑`. The former both passes the type decoration as the inherited type and returns it as the synthesised type. The latter takes the synthesised type and the inherited type and confirms they are identical — it should remind you of the equality test in the application rule in the first section.

Exercise `bidirectional-mul` (recommended)

Rewrite your definition of multiplication from Chapter Lambda, decorated to support inference.

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

Exercise `bidirectional-products` (recommended)

Extend the bidirectional type rules to include products from Chapter More.

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

Exercise `bidirectional-rest` (stretch)

Extend the bidirectional type rules to include the rest of the constructs from Chapter More.

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

Prerequisites

The rule for `M ↑` requires the ability to decide whether two types are equal. It is straightforward to code:

```
_≟Tp_ : (A B : Type) → Dec (A ≡ B)
`ℕ ≟Tp `ℕ                      = yes refl
`ℕ ≟Tp (A ⇒ B)                 = no λ()
(A ⇒ B) ≟Tp `ℕ                 = no λ()
(A ⇒ B) ≟Tp (A′ ⇒ B′)
  with A ≟Tp A′ | B ≟Tp B′
... | no A≢ | _                = no λ{refl → A≢ refl}
... | yes _ | no B≢         = no λ{refl → B≢ refl}
... | yes refl | yes refl = yes refl
```

We will also need a couple of obvious lemmas; the domain and range of equal function types are equal:

```
dom≡ : ∀ {A A′ B B′} → A ⇒ B ≡ A′ ⇒ B′ → A ≡ A′
dom≡ refl = refl

rng≡ : ∀ {A A′ B B′} → A ⇒ B ≡ A′ ⇒ B′ → B ≡ B′
rng≡ refl = refl
```

We will also need to know that the types `` `ℕ `` and `A ⇒ B` are not equal:

```
ℕ≢⇒ : ∀ {A B} → `ℕ ≢ A ⇒ B
ℕ≢⇒ ()
```

Unique types

Looking up a type in the context is unique. Given two derivations, one showing `Γ ∋ x ⦂ A` and one showing `Γ ∋ x ⦂ B`, it follows that `A` and `B` must be identical:

```
uniq-∋ : ∀ {Γ x A B} → Γ ∋ x ⦂ A → Γ ∋ x ⦂ B → A ≡ B
uniq-∋ Z Z                 = refl
uniq-∋ Z (S x≢y _)         = ⊥-elim (x≢y refl)
uniq-∋ (S x≢y _) Z         = ⊥-elim (x≢y refl)
uniq-∋ (S _ ∋x) (S _ ∋x′) = uniq-∋ ∋x ∋x′
```

If both derivations are by rule `Z` then uniqueness follows immediately, while if both derivations are by rule `S` then uniqueness follows by induction. It is a contradiction if one derivation is by rule `Z` and one by rule `S`, since rule `Z` requires the variable we are looking for is the final one in the context, while rule `S` requires it is not.

Synthesizing a type is also unique. Given two derivations, one showing `Γ ⊢ M ↑ A` and one showing `Γ ⊢ M ↑ B`, it follows that `A` and `B` must be identical:

```
uniq-↑ : ∀ {Γ M A B} → Γ ⊢ M ↑ A → Γ ⊢ M ↑ B → A ≡ B
uniq-↑ (⊢` ∋x) (⊢` ∋x′)       = uniq-∋ ∋x ∋x′
uniq-↑ (⊢L · ⊢M) (⊢L′ · ⊢M′) = rng≡ (uniq-↑ ⊢L ⊢L′)
uniq-↑ (⊢↓ ⊢M) (⊢↓ ⊢M′)       = refl
```

There are three possibilities for the term. If it is a variable, uniqueness of synthesis follows from uniqueness of lookup. If it is an application, uniqueness follows by induction on the function in the application, since the range of equal types are equal. If it is a switch expression, uniqueness follows since both terms are decorated with the same type.

Lookup type of a variable in the context

Given `Γ` and two distinct variables `x` and `y`, if there is no type `A` such that `Γ ∋ x ⦂ A` holds, then there is also no type `A` such that `Γ , y ⦂ B ∋ x ⦂ A` holds:

```
ext∋ : ∀ {Γ B x y}
  → x ≢ y
  → ¬ ∃[ A ]( Γ ∋ x ⦂ A )
    -----------------------------
  → ¬ ∃[ A ]( Γ , y ⦂ B ∋ x ⦂ A )
ext∋ x≢y _ ⟨ A , Z ⟩     = x≢y refl
ext∋ _ ¬∃ ⟨ 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 (ext∋ 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 `ext∋` 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 `⊢L` is evidence that `` Γ ⊢ L ⦂ `ℕ ``. Evidence that `Γ ⊢ L · M ↑ _` holds must have the form `⊢L′ · _` where `⊢L′` is evidence that `Γ ⊢ L ↑ A ⇒ B` for some types `A` and `B`. Applying `uniq-↑` to `⊢L` and `⊢L′` yields a contradiction, since `` `ℕ `` cannot equal `A ⇒ B`.
    * The other is that `⊢L` is evidence that `Γ ⊢ L ↑ A ⇒ B`, in which case we recurse on the argument `M`:
      · If it fails, then `¬⊢M` is evidence that `Γ ⊢ M ↓ A` does not hold. By `¬arg` applied to `⊢L` and `¬⊢M`, it follows that `Γ ⊢ L · M ↑ B` cannot hold.
      · If it succeeds, then `⊢M` is evidence that `Γ ⊢ M ↓ A`, and `⊢L · ⊢M` provides evidence that `Γ ⊢ L · M ↑ B`.
- If the term is a switch `M ↓ A` from synthesised to inherited, we recurse on the subterm `M`, supplying type `A` by inheritance:
  - If it fails, then `¬⊢M` is evidence that `Γ ⊢ M ↓ A` does not hold. Evidence that `Γ ⊢ (M ↓ A) ↑ A` holds must have the form `⊢↓ ⊢M` where `⊢M` is evidence that `Γ ⊢ M ↓ A` holds, which yields a contradiction.
  - If it succeeds, then `⊢M` is evidence that `Γ ⊢ M ↓ A`, and `⊢↓ ⊢M` provides evidence that `Γ ⊢ (M ↓ A) ↑ A`.

We next consider the code for inheritance:

```
inherit Γ (ƛ x ⇒ N) `ℕ      = no (λ())
inherit Γ (ƛ x ⇒ N) (A ⇒ B) with inherit (Γ , x ⦂ A) N B
... | no ¬⊢N                = no (λ{ (⊢ƛ ⊢N) → ¬⊢N ⊢N })
... | yes ⊢N                = yes (⊢ƛ ⊢N)
inherit Γ `zero `ℕ          = yes ⊢zero
inherit Γ `zero (A ⇒ B)     = no (λ())
inherit Γ (`suc M) `ℕ with inherit Γ M `ℕ
... | no ¬⊢M                = no (λ{ (⊢suc ⊢M) → ¬⊢M ⊢M })
... | yes ⊢M                = yes (⊢suc ⊢M)
```

```
inherit Γ (`suc M) (A ⇒ B) = no (λ())
inherit Γ (`case L [zero⇒ M |suc x ⇒ N ]) A with synthesize Γ L
... | no ¬∃                 = no (λ{ (⊢case ⊢L _ _) → ¬∃ ⟨ `ℕ , ⊢L ⟩})
... | yes ⟨ _ ⇒ _ , ⊢L ⟩    = no (λ{ (⊢case ⊢L′ _ _) → ℕ≢⇒ (uniq-↑ ⊢L′ ⊢L) })
... | yes ⟨ `ℕ , ⊢L ⟩ with inherit Γ M A
... | no ¬⊢M                = no (λ{ (⊢case _ ⊢M _) → ¬⊢M ⊢M })
... | yes ⊢M with inherit (Γ , x ⦂ `ℕ) N A
...     | no ¬⊢N            = no (λ{ (⊢case _ _ ⊢N) → ¬⊢N ⊢N })
...     | yes ⊢N            = yes (⊢case ⊢L ⊢M ⊢N)
inherit Γ (μ x ⇒ N) A with inherit (Γ , x ⦂ A) N A
... | no ¬⊢N                = no (λ{ (⊢μ ⊢N) → ¬⊢N ⊢N })
... | yes ⊢N                = yes (⊢μ ⊢N)
inherit Γ (M ↑) B with synthesize Γ M
... | no ¬∃                 = no (λ{ (⊢↑ ⊢M _) → ¬∃ ⟨ _ , ⊢M ⟩ })
... | yes ⟨ A , ⊢M ⟩ with A ≟Tp B
... | no A≢B                = no (¬switch ⊢M A≢B)
... | yes A≡B               = yes (⊢↑ ⊢M A≡B)
```

We consider only the cases for abstraction and and for switching from inherited to synthesized:

- If the term is an abstraction `ƛ x ⇒ N` and the inherited type is `` `ℕ ``, then it is trivial that `` Γ ⊢ (ƛ x ⇒ N) ↓ `ℕ `` cannot hold.

- If the term is an abstraction `ƛ x ⇒ N` and the inherited type is `A ⇒ B`, then we recurse with context `Γ , x ⦂ A` on subterm `N` inheriting type `B`:

  - If it fails, then `¬⊢N` is evidence that `Γ , x ⦂ A ⊢ N ↓ B` does not hold. Evidence that `Γ ⊢ (ƛ x ⇒ N) ↓ A ⇒ B` holds must have the form `⊢ƛ ⊢N` where `⊢N` is evidence that `Γ , x ⦂ A ⊢ N ↓ B`, which yields a contradiction.
  - If it succeeds, then `⊢N` is evidence that `Γ , x ⦂ A ⊢ N ↓ B` holds, and `⊢ƛ ⊢N` provides evidence that `Γ ⊢ (ƛ x ⇒ N) ↓ A ⇒ B`.

- If the term is a switch `M ↑` from inherited to synthesised, we recurse on the subterm `M`:

  - If it fails, then `¬∃` is evidence there is no `A` such that `Γ ⊢ M ↑ A` holds. Evidence that `Γ ⊢ (M ↑) ↓ B` holds must have the form `⊢↑ ⊢M _` where `⊢M` is evidence that `Γ ⊢ M ↑ _`, which yields a contradiction.
  - If it succeeds, then `⊢M` is evidence that `Γ ⊢ M ↑ A` holds. We apply `_≟Tp_` do decide whether `A` and `B` are equal:
    * If it fails, then `A≢B` is evidence that `A ≢ B`. By `¬switch` applied to `⊢M` and `A≢B` it follow that `Γ ⊢ (M ↑) ↓ B` cannot hold.
    * If it succeeds, then `A≡B` is evidence that `A ≡ B`, and `⊢↑ ⊢M A≡B` provides evidence that `Γ ⊢ (M ↑) ↓ B`.

The remaining cases are similar, and their code can pretty much be read directly from the corresponding typing rules.

Testing the example terms

First, we copy a function introduced earlier that makes it easy to compute the evidence that two variable names are distinct:

```
_≠_ : ∀ (x y : Id) → x ≢ y
x ≠ y with x ≟ y
... | no  x≢y = x≢y
... | yes _   = ⊥-elim impossible
  where postulate impossible : ⊥
```

Here is the result of typing two plus two on naturals:

```
⊢2+2 : ∅ ⊢ 2+2 ↑ `ℕ
⊢2+2 =
 (⊢↓
  (⊢μ
   (⊢ƛ
    (⊢ƛ
     (⊢case (⊢` (S ("m" ≠ "n") Z)) (⊢↑ (⊢` Z) refl)
      (⊢suc
       (⊢↑
        (⊢`
         (S ("p" ≠ "m")
          (S ("p" ≠ "n")
           (S ("p" ≠ "m") Z)))
         · ⊢↑ (⊢` Z) refl
         · ⊢↑ (⊢` (S ("n" ≠ "m") Z)) refl)
        refl))))))
  · ⊢suc (⊢suc ⊢zero)
  · ⊢suc (⊢suc ⊢zero))
```

We confirm that synthesis on the relevant term returns natural as the type and the above derivation:

```
_ : synthesize ∅ 2+2 ≡ yes ⟨ `ℕ , ⊢2+2 ⟩
_ = refl
```

Indeed, the above derivation was computed by evaluating the term on the left, with minor editing of the result. The only editing required was to replace Agda's representation of the evidence that two strings are unequal (which it cannot print nor read) by equivalent calls to `_≠_`.

Here is the result of typing two plus two with Church numerals:

```
⊢2+2ᶜ : ∅ ⊢ 2+2ᶜ ↑ `ℕ
⊢2+2ᶜ =
 ⊢↓
 (⊢ƛ
  (⊢ƛ
   (⊢ƛ
    (⊢ƛ
     (⊢↑
      (⊢`
       (S ("m" ≠ "z")
        (S ("m" ≠ "s")
         (S ("m" ≠ "n") Z)))
       · ⊢↑ (⊢` (S ("s" ≠ "z") Z)) refl
       ·
       ⊢↑
       (⊢`
        (S ("n" ≠ "z")
         (S ("n" ≠ "s") Z))
        · ⊢↑ (⊢` (S ("s" ≠ "z") Z)) refl
        · ⊢↑ (⊢` Z) refl)
       refl)
      refl)))))
 ·
 ⊢ƛ
 (⊢ƛ
  (⊢↑
   (⊢` (S ("s" ≠ "z") Z) ·
    ⊢↑ (⊢` (S ("s" ≠ "z") Z) · ⊢↑ (⊢` Z) refl)
    refl)
   refl))
 ·
 ⊢ƛ
 (⊢ƛ
  (⊢↑
   (⊢` (S ("s" ≠ "z") Z) ·
    ⊢↑ (⊢` (S ("s" ≠ "z") Z) · ⊢↑ (⊢` Z) refl)
    refl)
   refl))
 · ⊢ƛ (⊢suc (⊢↑ (⊢` Z) refl))
 · ⊢zero
```

We confirm that synthesis on the relevant term returns natural as the type and the above derivation:

```
_ : synthesize ∅ 2+2ᶜ ≡ yes ⟨ `ℕ , ⊢2+2ᶜ ⟩
_ = refl
```

Again, the above derivation was computed by evaluating the term on the left and editing.

Testing the error cases

It is important not just to check that code works as intended, but also that it fails as intended. Here are checks for several possible errors:

Unbound variable:

```
_ : synthesize ∅ ((ƛ "x" ⇒ ` "y" ↑) ↓ (`ℕ ⇒ `ℕ)) ≡ no _
_ = refl
```

Argument in application is ill typed:

```
_ : synthesize ∅ (plus · sucᶜ) ≡ no _
_ = refl
```

Function in application is ill typed:

```
_ : synthesize ∅ (plus · sucᶜ · two) ≡ no _
_ = refl
```

Function in application has type natural:

```
_ : synthesize ∅ ((two ↓ `ℕ) · two) ≡ no _
_ = refl
```

Abstraction inherits type natural:

```
_ : synthesize ∅ (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 (twoᶜ ↓ Ch) [zero⇒ `zero |suc "x" ⇒ ` "x" ↑ ] ↓ `ℕ) ) ≡ no _
_ = refl
```

Case of an ill-typed term:

```
_ : synthesize ∅
      ((`case (twoᶜ ↓ `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 ƛ_
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 x. x x)((λ x. x) a) —→ (λ x. x x) a
(λ x. x x)((λ x. x) a) —→ ((λ x. x) a) ((λ x. x) a)
```

Both terms can reduce to `a a`, but the second term requires two steps to get there, not one.

To side-step this problem, we'll define an auxiliary reduction relation, called *parallel reduction*, that can perform many reductions simultaneously and thereby satisfy the diamond property. Furthermore, we show that a parallel reduction sequence exists between any two terms if and only if a beta reduction sequence exists between them. Thus, we can reduce the proof of confluence for beta reduction to confluence for parallel reduction.

Imports

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Function using (_∘_)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import plfa.part2.Substitution using (Rename; Subst)
open import plfa.part2.Untyped
  using (_—→_; β; ξ₁; ξ₂; ζ; _—↠_; begin_; _—→⟨_⟩_; _—↠⟨_⟩_; _∎;
  abs-cong; appL-cong; appR-cong; —↠-trans;
  _⊢_; _∋_; `_; #_; _,_; ★; ƛ_; _·_; _[_];
  rename; ext; exts; Z; S_; subst; subst-zero)
```

Parallel Reduction

The parallel reduction relation is defined as follows.

```
infix 2 _⇛_

data _⇛_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  pvar : ∀{Γ A}{x : Γ ∋ A}
      ---------
    → (` x) ⇛ (` x)

  pabs : ∀{Γ}{N N′ : Γ , ★ ⊢ ★}
    → N ⇛ N′
      ----------
    → ƛ N ⇛ ƛ N′

  papp : ∀{Γ}{L L′ M M′ : Γ ⊢ ★}
    → L ⇛ L′
    → M ⇛ M′
      -----------------
    → L · M ⇛ L′ · M′

  pbeta : ∀{Γ}{N N′ : Γ , ★ ⊢ ★}{M M′ : Γ ⊢ ★}
    → N ⇛ N′
    → M ⇛ M′
      -----------------------
    → (ƛ N) · M ⇛ N′ [ M′ ]
```

The first three rules are congruences that reduce each of their parts simultaneously. The last rule reduces a lambda term and term in parallel followed by a beta step.

We remark that the `pabs`, `papp`, and `pbeta` rules perform reduction on all their subexpressions simultaneously. Also, the `pabs` rule is akin to the `ζ` rule and `pbeta` is akin to `β`.

Parallel reduction is reflexive.

```
par-refl : ∀{Γ A}{M : Γ ⊢ A} → M ⇛ M
par-refl {Γ} {A} {` x} = pvar
par-refl {Γ} {★} {ƛ N} = pabs par-refl
par-refl {Γ} {★} {L · M} = papp par-refl par-refl
```

We define the sequences of parallel reduction as follows.

```
infix 2 _⇛*_
infixr 2 _⇛⟨_⟩_
infix 3 _∎

data _⇛*_ : ∀ {Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where

  _∎ : ∀ {Γ A} (M : Γ ⊢ A)
       --------
    → M ⇛* M

  _⇛⟨_⟩_ : ∀ {Γ A} (L : Γ ⊢ A) {M N : Γ ⊢ A}
    → L ⇛ M
    → M ⇛* N
       ---------
    → L ⇛* N
```

Exercise `par-diamond-eg` **(practice)**

Revisit the counter example to the diamond property for reduction by showing that the diamond property holds for parallel reduction in that case.

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

Equivalence between parallel reduction and reduction

Here we prove that for any `M` and `N`, `M ⇛* N` if and only if `M —↠ N`. The only-if direction is particularly easy. We start by showing that if `M —→ N`, then `M ⇛ N`. The proof is by induction on the reduction `M —→ N`.

```
beta-par : ∀{Γ A}{M N : Γ ⊢ A}
  → M —→ N
    ------
  → M ⇛ N
beta-par {Γ} {★} {L · M} (ξ₁ r) = papp (beta-par {M = L} r) par-refl
beta-par {Γ} {★} {L · M} (ξ₂ r) = papp par-refl (beta-par {M = M} r)
beta-par {Γ} {★} {(ƛ N) · M} β = pbeta par-refl par-refl
beta-par {Γ} {★} {ƛ N} (ζ r) = pabs (beta-par r)
```

With this lemma in hand we complete the only-if direction, that `M —↠ N` implies `M ⇛* N`. The proof is a straightforward induction on the reduction sequence `M —↠ N`.

```
betas-pars : ∀{Γ A} {M N : Γ ⊢ A}
  → M —↠ N
    ------
  → M ⇛* N
betas-pars {Γ} {A} {M₁} {.M₁} (M₁ ∎) = M₁ ∎
betas-pars {Γ} {A} {.L} {N} (L —→⟨ b ⟩ bs) =
  L ⇛⟨ beta-par b ⟩ betas-pars bs
```

Now for the other direction, that `M ⇛* N` implies `M —↠ N`. The proof of this direction is a bit different because it's not the case that `M ⇛ N` implies `M —→ N`. After all, `M ⇛ N` performs many reductions. So instead we shall prove that `M ⇛ N` implies `M —↠ N`.

```
par-betas : ∀{Γ A}{M N : Γ ⊢ A}
  → M ⇛ N
    ------
  → M —↠ N
par-betas {Γ} {A} {.(` _)} (pvar{x = x}) = (` x) ∎
par-betas {Γ} {★} {ƛ N} (pabs p) = abs-cong (par-betas p)
par-betas {Γ} {★} {L · M} (papp {L = L}{L′}{M}{M′} p₁ p₂) =
    begin
    L · M  —↠⟨ appL-cong{M = M} (par-betas p₁) ⟩
    L′ · M —↠⟨ appR-cong (par-betas p₂) ⟩
    L′ · M′
    ∎
par-betas {Γ} {★} {(ƛ N) · M} (pbeta{N′ = N′}{M′ = M′} p₁ p₂) =
    begin
    (ƛ N) · M   —↠⟨ appL-cong{M = M} (abs-cong (par-betas p₁)) ⟩
    (ƛ N′) · M  —↠⟨ appR-cong{L = ƛ N′} (par-betas p₂) ⟩
    (ƛ N′) · M′ —→⟨ β ⟩
      N′ [ M′ ]
    ∎
```

The proof is by induction on `M ⇛ N`.

- Suppose `x ⇛ x`. We immediately have `x —↠ x`.
- Suppose `ƛ N ⇛ ƛ N′` because `N ⇛ N′`. By the induction hypothesis we have `N —↠ N′`. We conclude that `ƛ N —↠ ƛ N′` because `—↠` is a congruence.
- Suppose `L · M ⇛ L′ · M′` because `L ⇛ L′` and `M ⇛ M′`. By the induction hypothesis, we have `L —↠ L′` and `M —↠ M′`. So `L · M —↠ L′ · M` and then `L′ · M —↠ L′ · M′` because `—↠` is a congruence.
- Suppose `(ƛ N) · M ⇛ N′ [ M′ ]` because `N ⇛ N′` and `M ⇛ M′`. By similar reasoning, we have `(ƛ N) · M —↠ (ƛ N′) · M′` which we can following with the β reduction `(ƛ N′) · M′ —→ N′ [ M′ ]`.

With this lemma in hand, we complete the proof that `M ⇛* N` implies `M —↠ N` with a simple induction on `M ⇛* N`.

```
pars-betas : ∀{Γ A} {M N : Γ ⊢ A}
  → M ⇛* N
    ------
  → M —↠ N
pars-betas (M₁ ∎) = M₁ ∎
pars-betas (L ⇛⟨ p ⟩ ps) = —↠-trans (par-betas p) (pars-betas ps)
```

Substitution lemma for parallel reduction

Our next goal is the prove the diamond property for parallel reduction. But to do that, we need to prove that substitution respects parallel reduction. That is, if `N ⇛ N′` and `M ⇛ M′`, then `N [ M ] ⇛ N′ [ M′ ]`. We cannot prove this directly by induction, so we generalize it to: if `N ⇛ N′` and the substitution `σ` pointwise parallel reduces to `τ`, then `subst σ N ⇛ subst τ N′`. We define the notion of pointwise parallel reduction as follows.

```
par-subst : ∀{Γ Δ} → Subst Γ Δ → Subst Γ Δ → Set
par-subst {Γ}{Δ} σ σ′ = ∀{A}{x : Γ ∋ A} → σ x ⇛ σ′ x
```

Because substitution depends on the extension function `exts`, which in turn relies on `rename`, we start with a version of the substitution lemma, called `par-rename`, that is specialized to renamings. The proof of `par-rename` relies on the fact that renaming and substitution commute with one another, which is a lemma that we import from Chapter Substitution and restate here.

```
rename-subst-commute : ∀{Γ Δ}{N : Γ , ★ ⊢ ★}{M : Γ ⊢ ★}{ρ : Rename Γ Δ }
  → (rename (ext ρ) N) [ rename ρ M ] ≡ rename ρ (N [ M ])
rename-subst-commute {N = N} = plfa.part2.Substitution.rename-subst-commute {N = N}
```

Now for the `par-rename` lemma.

```
par-rename : ∀{Γ Δ A} {ρ : Rename Γ Δ} {M M′ : Γ ⊢ A}
  → M ⇛ M′
    ------------------------
  → rename ρ M ⇛ rename ρ M′
par-rename pvar = pvar
par-rename (pabs p) = pabs (par-rename p)
par-rename (papp p₁ p₂) = papp (par-rename p₁) (par-rename p₂)
par-rename {Γ}{Δ}{A}{ρ} (pbeta{Γ}{N}{N′}{M}{M′} p₁ p₂)
    with pbeta (par-rename{ρ = ext ρ} p₁) (par-rename{ρ = ρ} p₂)
... | G rewrite rename-subst-commute{Γ}{Δ}{N′}{M′}{ρ} = G
```

The proof is by induction on `M ⇛ M′`. The first four cases are straightforward so we just consider the last one for `pbeta`.

- Suppose `(ƛ N) · M ⇛ N′ [ M′ ]` because `N ⇛ N′` and `M ⇛ M′`. By the induction hypothesis, we have `rename (ext ρ) N ⇛ rename (ext ρ) N′` and `rename ρ M ⇛ rename ρ M′`. So by `pbeta` we have `(ƛ rename (ext ρ) N) · (rename ρ M) ⇛ (rename (ext ρ) N) [ rename ρ M ]`. However, to conclude we instead need parallel reduction to `rename ρ (N [ M ])`. But thankfully, renaming and substitution commute with one another.

With the `par-rename` lemma in hand, it is straightforward to show that extending substitutions preserves the pointwise parallel reduction relation.

```
par-subst-exts : ∀{Γ Δ} {σ τ : Subst Γ Δ}
  → par-subst σ τ
    ------------------------------------------
  → ∀{B} → par-subst (exts σ {B = B}) (exts τ)
par-subst-exts s {x = Z} = pvar
par-subst-exts s {x = S x} = par-rename s
```

The next lemma that we need for proving that substitution respects parallel reduction is the following which states that simultaneoous substitution commutes with single substitution. We import this lemma from Chapter Substitution and restate it below.

```
subst-commute : ∀{Γ Δ}{N : Γ , ★ ⊢ ★}{M : Γ ⊢ ★}{σ : Subst Γ Δ }
  → subst (exts σ) N [ subst σ M ] ≡ subst σ (N [ M ])
subst-commute {N = N} = plfa.part2.Substitution.subst-commute {N = N}
```

We are ready to prove that substitution respects parallel reduction.

```
subst-par : ∀{Γ Δ A} {σ τ : Subst Γ Δ} {M M′ : Γ ⊢ A}
  → par-subst σ τ → M ⇛ M′
    --------------------------
  → subst σ M ⇛ subst τ M′
subst-par {Γ} {Δ} {A} {σ} {τ} {` x} s pvar = s
subst-par {Γ} {Δ} {A} {σ} {τ} {ƛ N} s (pabs p) =
  pabs (subst-par {σ = exts σ} {τ = exts τ}
        (λ {A}{x} → par-subst-exts s {x = x}) p)
subst-par {Γ} {Δ} {★} {σ} {τ} {L · M} s (papp p₁ p₂) =
  papp (subst-par s p₁) (subst-par s p₂)
subst-par {Γ} {Δ} {★} {σ} {τ} {(ƛ N) · M} s (pbeta{N′ = N′}{M′ = M′} p₁ p₂)
    with pbeta (subst-par{σ = exts σ}{τ = exts τ}{M = N}
                 (λ{A}{x} → par-subst-exts s {x = x}) p₁)
               (subst-par {σ = σ} s p₂)
... | G rewrite subst-commute{N = N′}{M = M′}{σ = τ} = G
```

We proceed by induction on `M ⇛ M′`.

- Suppose `x ⇛ x`. We conclude that `σ x ⇛ τ x` using the premise `par-subst σ τ`.
- Suppose `ƛ N ⇛ ƛ N′` because `N ⇛ N′`. To use the induction hypothesis, we need `par-subst (exts σ) (exts τ)`, which we obtain by `par-subst-exts`. So we have `subst (exts σ) N ⇛ subst (exts τ) N′` and conclude by rule `pabs`.
- Suppose `L · M ⇛ L′ · M′` because `L ⇛ L′` and `M ⇛ M′`. By the induction hypothesis we have `subst σ L ⇛ subst τ L′` and `subst σ M ⇛ subst τ M′`, so we conclude by rule `papp`.
- Suppose `(ƛ N) · M ⇛ N′ [ M′ ]` because `N ⇛ N′` and `M ⇛ M′`. Again we obtain `par-subst (exts σ) (exts τ)` by `par-subst-exts`. So by the induction hypothesis, we have `subst (exts σ) N ⇛ subst (exts τ) N′` and `subst σ M ⇛ subst τ M′`. Then by rule `pbeta`, we have parallel reduction to `subst (exts τ) N′ [ subst τ M′ ]`. Substitution commutes with itself in the following sense. For any σ, N, and M, we have

  ```
      (subst (exts σ) N) [ subst σ M ] ≡ subst σ (N [ M ])
  ```

  So we have parallel reduction to `subst τ (N′ [ M′ ])`.

Of course, if `M ⇛ M′`, then `subst-zero M` pointwise parallel reduces to `subst-zero M′`.

```
par-subst-zero : ∀{Γ}{A}{M M′ : Γ ⊢ A}
  → M ⇛ M′
  → par-subst (subst-zero M) (subst-zero M′)
par-subst-zero {M} {M′} p {A} {Z} = p
par-subst-zero {M} {M′} p {A} {S x} = pvar
```

We conclude this section with the desired corollary, that substitution respects parallel reduction.

```
sub-par : ∀{Γ A B} {N N′ : Γ , A ⊢ B} {M M′ : Γ ⊢ A}
  → N ⇛ N′
  → M ⇛ M′
    --------------------------
  → N [ M ] ⇛ N′ [ M′ ]
sub-par pn pm = subst-par (par-subst-zero pm) pn
```

Parallel reduction satisfies the diamond property

The heart of the confluence proof is made of stone, or rather, of diamond! We show that parallel reduction satisfies the diamond property: that if `M ⇛ N` and `M ⇛ N′`, then `N ⇛ L` and `N′ ⇛ L` for some `L`. The typical proof is an induction on `M ⇛ N` and `M ⇛ N′` so that every possible pair gives rise to a witness `L` given by performing enough beta reductions in parallel.

However, a simpler approach is to perform as many beta reductions in parallel as possible on `M`, say `M ⁺`, and then show that `N` also parallel reduces to `M ⁺`. This is the idea of Takahashi's *complete development*. The desired property may be illustrated as

```
    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 `(λ M) ⁺ = λ (M ⁺)`, so we conclude that `λ N ⇛ λ (M ⁺)`.
- Suppose `(λ 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 `((λ N) · M) ⁺`, hence `N′ [ M′ ] ⇛ ((λ N) · M) ⁺`.
- Suppose `(λ L) · M ⇛ (λ L′) · M′`. By the induction hypothesis we have `L′ ⇛ L ⁺` and `M′ ⇛ M ⁺`; by definition `((λ L) · M) ⁺ = L ⁺ [ M ⁺ ]`. It follows `(λ 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.

```
    M
   /|\
  / | \
 /  |  \
N   2   N′
 \  |  /
  \ | /
   \|/
    M⁺
```

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:

```
        L
       / \
      1   *
     /     \
    M₁ (a)  M₂
   / \     /
  *   *   1
 /     \ /
M₁′(b)  N
 \     /
  *   *
   \ /
```

```
    N′
```

where downward lines are instances of `⇛` or `⇛*`, depending on how they are marked. Here `(a)` holds by `strip` and `(b)` holds by induction.

Proof of confluence for reduction

Confluence of reduction is a corollary of confluence for parallel reduction. From `L —↠ M₁` and `L —↠ M₂` we have `L ⇛* M₁` and `L ⇛* M₂` by `betas-pars`. Then by confluence we obtain some `L` such that `M₁ ⇛* N` and `M₂ ⇛* N`, from which we conclude that `M₁ —↠ N` and `M₂ —↠ N` by `pars-betas`.

```
confluence : ∀{Γ A} {L M₁ M₂ : Γ ⊢ A}
  → L —↠ M₁
  → L —↠ M₂
    -----------------------------------
  → Σ[ N ∈ Γ ⊢ A ] (M₁ —↠ N) × (M₂ —↠ N)
confluence L↠M₁ L↠M₂
    with par-confluence (betas-pars L↠M₁) (betas-pars L↠M₂)
... | ⟨ N , ⟨ M₁⇛N , M₂⇛N ⟩ ⟩ =
      ⟨ N , ⟨ pars-betas M₁⇛N , pars-betas M₂⇛N ⟩ ⟩
```

Notes

Broadly speaking, this proof of confluence, based on parallel reduction, is due to W. Tait and P. Martin-Löf (see Barendregt 1984, Section 3.2). Details of the mechanization come from several sources. The `subst-par` lemma is the "strong substitutivity" lemma of Shafer, Tebbi, and Smolka (ITP 2015). The proofs of `par-triangle`, `strip`, and `par-confluence` are based on the notion of complete development by Takahashi (1995) and Pfenning's 1992 technical report about the Church-Rosser theorem. In addition, we consulted Nipkow and Berghofer's mechanization in Isabelle, which is based on an earlier article by Nipkow (JAR 1996).

Unicode

This chapter uses the following unicode:

```
⇛  U+21DB  RIGHTWARDS TRIPLE ARROW (\r== or \Rrightarrow)
⁺  U+207A  SUPERSCRIPT PLUS SIGN   (\^+)
```

Chapter 19

BigStep: Big-step semantics of untyped lambda calculus

```
module plfa.part2.BigStep where
```

Introduction

The call-by-name evaluation strategy is a deterministic method for computing the value of a program in the lambda calculus. That is, call-by-name produces a value if and only if beta reduction can reduce the program to a lambda abstraction. In this chapter we define call-by-name evaluation and prove the forward direction of this if-and-only-if. The backward direction is traditionally proved via Curry-Feys standardisation, which is quite complex. We give a sketch of that proof, due to Plotkin, but postpone the proof in Agda until after we have developed a denotational semantics for the lambda calculus, at which point the proof is an easy corollary of properties of the denotational semantics.

We present the call-by-name strategy as a relation between an input term and an output value. Such a relation is often called a *big-step semantics*, written `M ⇓ V`, as it relates the input term `M` directly to the final result `V`, in contrast to the small-step reduction relation, `M —→ M′`, that maps `M` to another term `M′` in which a single sub-computation has been completed.

Imports

```
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; trans; sym; cong-app)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Function using (_∘_)
open import plfa.part2.Untyped
  using (Context; _⊢_; _∋_; ★; ∅; _,_; Z; S_; `_; #_; ƛ_; _·_;
  subst; subst-zero; exts; rename; β; ξ₁; ξ₂; ζ; _—→_; _—↠_; _—→⟨_⟩_; _∎;
  —↠-trans; appL-cong)
open import plfa.part2.Substitution using (Subst; ids)
```

Environments

To handle variables and function application, there is the choice between using substitution, as in `—→`, or to use an *environment*. An environment in call-by-name is a map from variables to closures, that is, to terms paired with their environments. We choose to use environments instead of substitution because the point of the call-by-name strategy is to be closer to an implementation of the language. Also, the denotational semantics introduced in later chapters uses environments and the proof of adequacy is made easier by aligning these choices.

We define environments and closures as follows.

```
ClosEnv : Context → Set

data Clos : Set where
  clos : ∀{Γ} → (M : Γ ⊢ ★) → ClosEnv Γ → Clos

ClosEnv Γ = ∀ (x : Γ ∋ ★) → Clos
```

As usual, we have the empty environment, and we can extend an environment.

```
∅' : ClosEnv ∅
∅' ()

_,'_ : ∀ {Γ} → ClosEnv Γ → Clos → ClosEnv (Γ , ★)
(γ ,' c) Z = c
(γ ,' c) (S x) = γ x
```

Big-step evaluation

The big-step semantics is represented as a ternary relation, written `γ ⊢ M ⇓ V`, where `γ` is the environment, `M` is the input term, and `V` is the result value. A *value* is a closure whose term is a lambda abstraction.

```
data _⊢_⇓_ : ∀{Γ} → ClosEnv Γ → (Γ ⊢ ★) → Clos → Set where

  ⇓-var : ∀{Γ}{γ : ClosEnv Γ}{x : Γ ∋ ★}{Δ}{δ : ClosEnv Δ}{M : Δ ⊢ ★}{V}
    → γ x ≡ clos M δ
    → δ ⊢ M ⇓ V
      -----------
    → γ ⊢ ` x ⇓ V

  ⇓-lam : ∀{Γ}{γ : ClosEnv Γ}{M : Γ , ★ ⊢ ★}
    → γ ⊢ ƛ M ⇓ clos (ƛ M) γ

  ⇓-app : ∀{Γ}{γ : ClosEnv Γ}{L M : Γ ⊢ ★}{Δ}{δ : ClosEnv Δ}{N : Δ , ★ ⊢ ★}{V}
    → γ ⊢ L ⇓ clos (ƛ N) δ → (δ ,' clos M γ) ⊢ N ⇓ V
      ---------------------------------------------------
    → γ ⊢ L · M ⇓ V
```

- The `⇓-var` rule evaluates a variable by finding the associated closure in the environment and then evaluating the closure.

- The `⇓-lam` rule turns a lambda abstraction into a closure by packaging it up with its environment.

- The `⇓-app` rule performs function application by first evaluating the term `L` in operator position. If that produces a closure containing a lambda abstraction `ƛ N`, then we evaluate the body `N` in an environment extended with the argument `M`. Note that `M` is not evaluated in rule `⇓-app` because this is call-by-name and not call-by-value.

Exercise `big-step-eg` (practice)

Show that `(ƛ ƛ # 1) · ((ƛ # 0 · # 0) · (ƛ # 0 · # 0))` terminates under big-step call-by-name evaluation.

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

The big-step semantics is deterministic

If the big-step relation evaluates a term `M` to both `V` and `V′`, then `V` and `V′` must be identical. In other words, the call-by-name relation is a partial function. The proof is a straightforward induction on the two big-step derivations.

```
⇓-determ : ∀{Γ}{γ : ClosEnv Γ}{M : Γ ⊢ ★}{V V' : Clos}
  → γ ⊢ M ⇓ V → γ ⊢ M ⇓ V'
  → V ≡ V'
⇓-determ (⇓-var eq1 mc) (⇓-var eq2 mc')
  with trans (sym eq1) eq2
... | refl = ⇓-determ mc mc'
⇓-determ ⇓-lam ⇓-lam = refl
⇓-determ (⇓-app mc mc₁) (⇓-app mc' mc'')
  with ⇓-determ mc mc'
... | refl = ⇓-determ mc₁ mc''
```

Big-step evaluation implies beta reduction to a lambda

If big-step evaluation produces a value, then the input term can reduce to a lambda abstraction by beta reduction:

```
  ∅' ⊢ M ⇓ clos (ƛ N′) δ
  -----------------------------
→ Σ[ N ∈ ∅ , ★ ⊢ ★ ] (M —↠ ƛ N)
```

The proof is by induction on the big-step derivation. As is often necessary, one must generalize the statement to get the induction to go through. In the case for `⇓-app` (function application), the argument is added to the environment, so the environment becomes non-empty. The corresponding β reduction substitutes the argument into the body of the lambda abstraction. So we generalize the lemma to allow an arbitrary environment `γ` and we add a premise that relates the environment `γ` to an equivalent substitution `σ`.

The case for `⇓-app` also requires that we strengthen the conclusion. In the case for `⇓-app` we have `γ ⊢ L ⇓ clos (λ N) δ` and the induction hypothesis gives us `L —↠ ƛ N′`, but we need to know that `N` and `N′` are equivalent. In particular, that `N′ ≡ subst τ N` where `τ` is the substitution that is equivalent to `δ`. Therefore we expand the conclusion of the statement, stating that the results are equivalent.

We make the two notions of equivalence precise by defining the following two mutually-recursive predicates `V ≈ M` and `γ ≈ₑ σ`.

```
_≈_ : Clos → (∅ ⊢ ★) → Set
_≈ₑ_ : ∀{Γ} → ClosEnv Γ → Subst Γ ∅ → Set

(clos {Γ} M γ) ≈ N = Σ[ σ ∈ Subst Γ ∅ ] γ ≈ₑ σ × (N ≡ subst σ M)

γ ≈ₑ σ = ∀{x} → (γ x) ≈ (σ x)
```

We can now state the main lemma:

```
If γ ⊢ M ⇓ V  and  γ ≈ₑ σ,
then  subst σ M —↠ N  and  V ≈ N  for some N.
```

Before starting the proof, we establish a couple lemmas about equivalent environments and substitutions.

The empty environment is equivalent to the identity substitution `ids`, which we import from Chapter Substitution.

```
≈ₑ-id : ∅' ≈ₑ ids
≈ₑ-id {()}
```

Of course, applying the identity substitution to a term returns the same term.

```
sub-id : ∀{Γ} {A} {M : Γ ⊢ A} → subst ids M ≡ M
sub-id = plfa.part2.Substitution.sub-id
```

We define an auxiliary function for extending a substitution.

```
ext-subst : ∀{Γ Δ} → Subst Γ Δ → Δ ⊢ ★ → Subst (Γ , ★) Δ
ext-subst{Γ}{Δ} σ N {A} = subst (subst-zero N) ∘ exts σ
```

The next lemma we need to prove states that if you start with an equivalent environment and substitution `γ ≈ₑ σ`, extending them with an equivalent closure and term `c ≈ N` produces an equivalent environment and substitution: `(γ ,' V) ≈ₑ (ext-subst σ N)`, or equivalently, `(γ ,' V) x ≈ₑ (ext-subst σ N) x` for any variable `x`. The proof will be by induction on `x` and for the induction step we need the following lemma, which states that applying the composition of `exts σ` and `subst-zero` to `S x` is the same as just `σ x`, which is a corollary of a theorem in Chapter Substitution.

```
subst-zero-exts : ∀{Γ Δ}{σ : Subst Γ Δ}{B}{M : Δ ⊢ B}{x : Γ ∋ ★}
  → (subst (subst-zero M) ∘ exts σ) (S x) ≡ σ x
subst-zero-exts {Γ}{Δ}{σ}{B}{M}{x} =
  cong-app (plfa.part2.Substitution.subst-zero-exts-cons{σ = σ}) (S x)
```

So the proof of `≈ₑ-ext` is as follows.

```
≈ₑ-ext : ∀ {Γ} {γ : ClosEnv Γ} {σ : Subst Γ ∅} {V} {N : ∅ ⊢ ★}
  → γ ≈ₑ σ → V ≈ N
    ---------------------------
  → (γ ,' V) ≈ₑ (ext-subst σ N)
```

```
≈ₑ-ext {Γ} {γ} {σ} {V} {N} γ≈ₑσ V≈N {Z} = V≈N
≈ₑ-ext {Γ} {γ} {σ} {V} {N} γ≈ₑσ V≈N {S x}
  rewrite subst-zero-exts {σ = σ}{M = N}{x} = γ≈ₑσ
```

We proceed by induction on the input variable.

- If it is `Z`, then we immediately conclude using the premise `V ≈ N`.
- If it is `S x`, then we rewrite using the `subst-zero-exts` lemma and use the premise `γ ≈ₑ σ` to conclude.

To prove the main lemma, we need another technical lemma about substitution. Applying one substitution after another is the same as composing the two substitutions and then applying them.

```
sub-sub : ∀{Γ Δ Σ}{A}{M : Γ ⊢ A} {σ₁ : Subst Γ Δ}{σ₂ : Subst Δ Σ}
  → subst σ₂ (subst σ₁ M) ≡ subst (subst σ₂ ∘ σ₁) M
sub-sub {M = M} = plfa.part2.Substitution.sub-sub {M = M}
```

We arive at the main lemma: if `M` big steps to a closure `V` in environment `γ`, and if `γ ≈ₑ σ`, then `subst σ M` reduces to some term `N` that is equivalent to `V`. We describe the proof below.

```
⇓→—↠×≈ : ∀{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
⇓→—↠×≈ {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 ⇓→—↠×≈{σ = τ} δ⊢L⇓V δ≈ₑτ
...   | ⟨ N , ⟨ τL—↠N , V≈N ⟩ ⟩ rewrite σx≡τL =
        ⟨ N , ⟨ τL—↠N , V≈N ⟩ ⟩
⇓→—↠×≈ {σ = σ} {V = clos (ƛ N) γ} (⇓-lam) γ≈ₑσ =
  ⟨ subst σ (ƛ N) , ⟨ subst σ (ƛ N) ∎ , ⟨ σ , ⟨ γ≈ₑσ , refl ⟩ ⟩ ⟩ ⟩
⇓→—↠×≈{Γ}{γ} {σ = σ} {L · M} {V} (⇓-app {N = N} L⇓ƛNδ N⇓V) γ≈ₑσ
  with ⇓→—↠×≈{σ = σ} L⇓ƛNδ γ≈ₑσ
... | ⟨ _ , ⟨ σL—↠ƛτN , ⟨ τ , ⟨ δ≈ₑτ , ≡ƛτN ⟩ ⟩ ⟩ ⟩ rewrite ≡ƛτN
      with ⇓→—↠×≈ {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 λᵥ 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 ⊢ (L · M) ↓ w

  ↦-intro : ∀ {Γ} {γ : Env Γ} {N v w}
    → γ `, v ⊢ N ↓ w
      -------------------
    → γ ⊢ (ƛ N) ↓ (v ↦ w)
  ⊥-intro : ∀ {Γ} {γ : Env Γ} {M}
      ---------
    → γ ⊢ M ↓ ⊥

  ⊔-intro : ∀ {Γ} {γ : Env Γ} {M v w}
    → γ ⊢ M ↓ v
    → γ ⊢ M ↓ w
      ---------------
    → γ ⊢ M ↓ (v ⊔ w)

  sub : ∀ {Γ} {γ : Env Γ} {M v w}
    → γ ⊢ M ↓ v
    → w ⊑ v
      ---------
    → γ ⊢ M ↓ w
```

Consider the rule for lambda abstractions, `↦-intro`. It says that a lambda abstraction results in a single-entry table that maps the input `v` to the output `w`, provided that evaluating the body in an environment with `v` bound to its parameter produces the output `w`. As a simple example of this rule, we can see that the identity function maps `⊥` to `⊥` and also that it maps `⊥ ↦ ⊥` to `⊥ ↦ ⊥`.

```
id : ∅ ⊢ ★
id = ƛ # 0
```

```
denot-id1 : ∀ {γ} → γ ⊢ id ↓ ⊥ ↦ ⊥
denot-id1 = ↦-intro var

denot-id2 : ∀ {γ} → γ ⊢ id ↓ (⊥ ↦ ⊥) ↦ (⊥ ↦ ⊥)
denot-id2 = ↦-intro var
```

Of course, we will need tables with many rows to capture the meaning of lambda abstractions. These can be constructed using the `⊔-intro` rule. If term M (typically a lambda abstraction) can produce both tables `v` and `w`, then it produces the combined table `v ⊔ w`. One can take an operational view of the rules `↦-intro` and `⊔-intro` by imagining that when an interpreter first comes to a lambda abstraction, it pre-evaluates the function on a bunch of randomly chosen arguments, using many instances of the rule `↦-intro`, and then joins them into a big table using many instances of the rule `⊔-intro`. In the following we show that the identity function produces a table containing both of the previous results, `⊥ ↦ ⊥` and `(⊥ ↦ ⊥) ↦ (⊥ ↦ ⊥)`.

```
denot-id3 : `∅ ⊢ id ↓ (⊥ ↦ ⊥) ⊔ (⊥ ↦ ⊥) ↦ (⊥ ↦ ⊥)
denot-id3 = ⊔-intro denot-id1 denot-id2
```

We most often think of the judgment `γ ⊢ M ↓ v` as taking the environment `γ` and term `M` as input, producing the result `v`. However, it is worth emphasizing that the semantics is a *relation*. The above results for the identity function show that the same environment and term can be mapped to different results. However, the results for a given `γ` and `M` are not *too* different, they are all finite approximations of the same function. Perhaps a better way of thinking about the judgment `γ ⊢ M ↓ v` is that the `γ`, `M`, and `v` are all inputs and the semantics either confirms or denies whether `v` is an accurate partial description of the result of `M` in environment `γ`.

Next we consider the meaning of function application as given by the `↦-elim` rule. In the premise of the rule we have that `L` maps `v` to `w`. So if `M` produces `v`, then the application of `L` to `M` produces `w`.

As an example of function application and the `↦-elim` rule, we apply the identity function to itself. Indeed, we have both that `∅ ⊢ id ↓ (u ↦ u) ↦ (u ↦ u)` and also `∅ ⊢ id ↓ (u ↦ u)`, so we can apply the rule `↦-elim`.

```
id-app-id : ∀ {u : Value} → `∅ ⊢ id · id ↓ (u ↦ u)
id-app-id {u} = ↦-elim (↦-intro var) (↦-intro var)
```

Next we revisit the Church numeral two: `λ f. λ u. (f (f u))`. This function has two parameters: a function `f` and an arbitrary value `u`, and it applies `f` twice. So `f` must map `u` to some value, which we'll name `v`. Then for the second application, `f` must map `v` to some value. Let's name it `w`. So the function's table must include two entries, both `u ↦ v` and `v ↦ w`. For each application of the table, we extract the appropriate entry from it using the `sub` rule. In particular, we use the ⊑-conj-R1 and ⊑-conj-R2 to select `u ↦ v` and `v ↦ w`, respectively, from the table `u ↦ v ⊔ v ↦ w`. So the meaning of two^{c} is that it takes this table and parameter `u`, and it returns `w`. Indeed we derive this as follows.

```
denot-twoᶜ : ∀{u v w : Value} → `∅ ⊢ twoᶜ ↓ ((u ↦ v ⊔ v ↦ w) ↦ u ↦ w)
denot-twoᶜ {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 `v₁ = v₂ = ⊥`). 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-plusᶜ` **(practice)**

What is a denotation for `plusᶜ`? That is, find a value `v` (other than `⊥`) such that `∅ ⊢ plusᶜ ↓ v`. Also, give the proof of `∅ ⊢ plusᶜ ↓ 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 ℰ gives this alternative view of the semantics, which really just amounts to changing the order of the parameters.

```
ℰ : ∀{Γ} → (M : Γ ⊢ ★) → Denotation Γ
ℰ M = λ γ v → γ ⊢ M ↓ v
```

In general, two denotations are equal when they produce the same values in the same environment.

```
infix 3 _≃_

_≃_ : ∀ {Γ} → (Denotation Γ) → (Denotation Γ) → Set
(_≃_ {Γ} D₁ D₂) = (γ : Env Γ) → (v : Value) → D₁ γ v iff D₂ γ v
```

Denotational equality is an equivalence relation.

```
≃-refl : ∀ {Γ : Context} → {M : Denotation Γ}
  → M ≃ M
≃-refl γ v = ⟨ (λ x → x) , (λ x → x) ⟩

≃-sym : ∀ {Γ : Context} → {M N : Denotation Γ}
  → M ≃ N
    -----
  → N ≃ M
≃-sym eq γ v = ⟨ (proj₂ (eq γ v)) , (proj₁ (eq γ v)) ⟩

≃-trans : ∀ {Γ : Context} → {M₁ M₂ M₃ : Denotation Γ}
  → M₁ ≃ M₂
  → M₂ ≃ M₃
    -------
  → M₁ ≃ M₃
≃-trans eq1 eq2 γ v = ⟨ (λ z → proj₁ (eq2 γ v) (proj₁ (eq1 γ v) z)) ,
                        (λ z → proj₂ (eq1 γ v) (proj₂ (eq2 γ v) z)) ⟩
```

Two terms `M` and `N` are denotational equal when their denotations are equal, that is, `ℰ M ≃ ℰ N`.

The following submodule introduces equational reasoning for the `≃` relation.

```
module ≃-Reasoning {Γ : Context} where

  infix 1 start_
  infixr 2 _≃⟨⟩_ _≃⟨_⟩_
  infix 3 _☐

  start_ : ∀ {x y : Denotation Γ}
    → x ≃ y
      -----
    → x ≃ y
  start x≃y = x≃y

  _≃⟨_⟩_ : ∀ (x : Denotation Γ) {y z : Denotation Γ}
    → x ≃ y
    → y ≃ z
      -----
    → x ≃ z
  (x ≃⟨ x≃y ⟩ y≃z) = ≃-trans x≃y y≃z

  _≃⟨⟩_ : ∀ (x : Denotation Γ) {y : Denotation Γ}
    → x ≃ y
      -----
    → x ≃ y
  x ≃⟨⟩ x≃y  = x≃y

  _☐ : ∀ (x : Denotation Γ)
      -----
    → x ≃ x
  (x ☐)  = ≃-refl
```

Road map for the following chapters

The subsequent chapters prove that the denotational semantics has several desirable properties. First, we prove that the semantics is compositional, i.e., that the denotation of a term is a function of the denotations of its subterms. To do this we shall prove equations of the following shape.

```
ℰ (` x) ≃ ...
ℰ (ƛ M) ≃ ... ℰ M ...
ℰ (M · N) ≃ ... ℰ M ... ℰ N ...
```

The compositionality property is not trivial because the semantics we have defined includes three rules that are not syntax directed: `⊥-intro`, `⊔-intro`, and `sub`. The above equations suggest that the denotational semantics can be defined as a recursive function, and indeed, we give such a definition and prove that it is equivalent to 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 ⨆codΓ⊆w′ = ⊆↦→⨆cod⊆ Γ⊆v34 in
  ⟨ lt1 u↦u′∈Γ , ⊑-trans lt2 (⊆→⊑ ⨆codΓ⊆w′) ⟩
```

Notes

The denotational semantics presented in this chapter is an example of a *filter model* (Barendregt, Coppo, Dezani-Ciancaglini, 1983). Filter models use type systems with intersection types to precisely characterize runtime behavior (Coppo, Dezani-Ciancaglini, and Salle, 1979). The notation that we use in this chapter is not that of type systems and intersection types, but the `Value` data type is isomorphic to types (`↦` is `→`, `⊔` is `∧`, `⊥` is `⊤`), the `⊑` relation is the inverse of subtyping

`<:`, and the evaluation relation `ρ ⊢ M ↓ v` is isomorphic to a type system. Write `Γ` instead of `ρ`, `A` instead of `v`, and replace `↓` with `:` and one has a typing judgement `Γ ⊢ M : A`. By varying the definition of subtyping and using different choices of type atoms, intersection type systems provide semantics for many different untyped λ calculi, from full beta to the lazy and call-by-value calculi (Alessi, Barbanera, and Dezani-Ciancaglini, 2006) (Rocca and Paolini, 2004). The denotational semantics in this chapter corresponds to the BCD system (Barendregt, Coppo, Dezani-Ciancaglini, 1983). Part 3 of the book *Lambda Calculus with Types* describes a framework for intersection type systems that enables results similar to the ones in this chapter, but for the entire family of intersection type systems (Barendregt, Dekkers, and Statman, 2013).

The two ideas of using finite tables to represent functions and of relaxing table lookup to enable self application first appeared in a technical report by Gordon Plotkin (1972) and are later described in an article in Theoretical Computer Science (Plotkin 1993). In that work, the inductive definition of `Value` is a bit different than the one we use:

```
Value = C + ℘f(Value) × ℘f(Value)
```

where `C` is a set of constants and `℘f` means finite powerset. The pairs in `℘f(Value) × ℘f(Value)` represent input-output mappings, just as in this chapter. The finite powersets are used to enable a function table to appear in the input and in the output. These differences amount to changing where the recursion appears in the definition of `Value`. Plotkin's model is an example of a *graph model* of the untyped lambda calculus (Barendregt, 1984). In a graph model, the semantics is presented as a function from programs and environments to (possibly infinite) sets of values. The semantics in this chapter is instead defined as a relation, but set-valued functions are isomorphic to relations. Indeed, we present the semantics as a function in the next chapter and prove that it is equivalent to the relational version.

Dana Scott's ℘(ω) (1976) and Engeler's B(A) (1981) are two more examples of graph models. Both use the following inductive definition of `Value`.

```
Value = C + ℘f(Value) × Value
```

The use of `Value` instead of `℘f(Value)` in the output does not restrict expressiveness compared to Plotkin's model because the semantics use sets of values and a pair of sets `(V, V′)` can be represented as a set of pairs `{ (V, v′) | v′ ∈ V′ }`. In Scott's ℘(ω), the above values are mapped to and from the natural numbers using a kind of Godel encoding.

References

- Intersection Types and Lambda Models. Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini, Theoretical Compututer Science, vol. 355, pages 108-126, 2006.

- The Lambda Calculus. H.P. Barendregt, 1984.

- A filter lambda model and the completeness of type assignment. Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini, Journal of Symbolic Logic, vol. 48, pages 931-940, 1983.

- Lambda Calculus with Types. Henk Barendregt, Wil Dekkers, and Richard Statman, Cambridge University Press, Perspectives in Logic,

2013.

- Functional characterization of some semantic equalities inside λ-calculus. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Salle, in Sixth Colloquium on Automata, Languages and Programming. Springer, pages 133–146, 1979.

- Algebras and combinators. Erwin Engeler, Algebra Universalis, vol. 13, pages 389-392, 1981.
- A Set-Theoretical Definition of Application. Gordon D. Plotkin, University of Edinburgh, Technical Report MIP-R-95, 1972.
- Set-theoretical and other elementary models of the λ-calculus. Gordon D. Plotkin, Theoretical Computer Science, vol. 121, pages 351-409, 1993.
- The Parametric Lambda Calculus. Simona Ronchi Della Rocca and Luca Paolini, Springer, 2004.
- Data Types as Lattices. Dana Scott, SIAM Journal on Computing, vol. 5, pages 522-587, 1976.

Unicode

This chapter uses the following unicode:

```
⊥  U+22A5  UP TACK (\bot)
↦  U+21A6  RIGHTWARDS ARROW FROM BAR (\mapsto)
⊔  U+2294  SQUARE CUP (\lub)
⊑  U+2291  SQUARE IMAGE OF OR EQUAL TO (\sqsubseteq)
⨆ U+2A06  N-ARY SQUARE UNION OPERATOR (\Lub)
⊢  U+22A2  RIGHT TACK (\|- or \vdash)
↓  U+2193  DOWNWARDS ARROW (\d)
ᶜ  U+1D9C  MODIFIER LETTER SMALL C (\^c)
ℰ  U+2130  SCRIPT CAPITAL E (\McE)
≃  U+2243  ASYMPTOTICALLY EQUAL TO (\~- or \simeq)
∈  U+2208  ELEMENT OF (\in)
⊆  U+2286  SUBSET OF OR EQUAL TO (\sub= or \subseteq)
```

Chapter 21

Compositional: The denotational semantics is compositional

```
module plfa.part3.Compositional where
```

Introduction

In this chapter we prove that the denotational semantics is compositional, which means we fill in the ellipses in the following equations.

```
ℰ (` x) ≃ ...
ℰ (ƛ M) ≃ ... ℰ M ...
ℰ (M · N) ≃ ... ℰ M ... ℰ N ...
```

Such equations would imply that the denotational semantics could be instead defined as a recursive function. Indeed, we end this chapter with such a definition and prove that it is equivalent to ℰ.

Imports

```
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Data.Unit using (⊤; tt)
open import plfa.part2.Untyped
  using (Context; _,_; ★; _∋_; _⊢_; `_; ƛ_; _·_)
open import plfa.part3.Denotational
  using (Value; _↦_; _`,_; _⊔_; ⊥; _⊑_; _⊢_↓_;
         ⊑-bot; ⊑-fun; ⊑-conj-L; ⊑-conj-R1; ⊑-conj-R2;
         ⊑-dist; ⊑-refl; ⊑-trans; ⊔↦⊔-dist;
         var; ↦-intro; ↦-elim; ⊔-intro; ⊥-intro; sub;
         up-env; ℰ; _≃_; ≃-sym; Denotation; Env)
open plfa.part3.Denotational.≃-Reasoning
```

Equation for lambda abstraction

Regarding the first equation

```
ℰ (ƛ M) ≃ ... ℰ M ...
```

we need to define a function that maps a `Denotation (Γ , ★)` to a `Denotation Γ`. This function, let us name it `ℱ`, should mimic the non-recursive part of the semantics when applied to a lambda term. In particular, we need to consider the rules `↦-intro`, `⊥-intro`, and `⊔-intro`. So `ℱ` has three parameters, the denotation `D` of the subterm `M`, an environment `γ`, and a value `v`. If we define `ℱ` by recursion on the value `v`, then it matches up nicely with the three rules `↦-intro`, `⊥-intro`, and `⊔-intro`.

```
ℱ : ∀{Γ} → Denotation (Γ , ★) → Denotation Γ
ℱ D γ (v ↦ w) = D (γ `, v) w
ℱ D γ ⊥ = ⊤
ℱ D γ (u ⊔ v) = (ℱ D γ u) × (ℱ D γ v)
```

If one squints hard enough, the `ℱ` function starts to look like the `curry` operation familiar to functional programmers. It turns a function that expects a tuple of length `n + 1` (the environment `Γ , ★`) into a function that expects a tuple of length `n` and returns a function of one parameter.

Using this `ℱ`, we hope to prove that

```
ℰ (ƛ N) ≃ ℱ (ℰ N)
```

The function `ℱ` is preserved when going from a larger value `v` to a smaller value `u`. The proof is a straightforward induction on the derivation of `u ⊑ v`, using the `up-env` lemma in the case for the `⊑-fun` rule.

```
sub-ℱ : ∀{Γ}{N : Γ , ★ ⊢ ★}{γ v u}
  → ℱ (ℰ N) γ v
  → u ⊑ v
    ------------
  → ℱ (ℰ N) γ u
sub-ℱ d ⊑-bot = tt
sub-ℱ d (⊑-fun lt lt′) = sub (up-env d lt) lt′
sub-ℱ d (⊑-conj-L lt lt₁) = ⟨ sub-ℱ d lt , sub-ℱ d lt₁ ⟩
sub-ℱ d (⊑-conj-R1 lt) = sub-ℱ (proj₁ d) lt
sub-ℱ d (⊑-conj-R2 lt) = sub-ℱ (proj₂ d) lt
sub-ℱ {v = v₁ ↦ v₂ ⊔ v₁ ↦ v₃} {v₁ ↦ (v₂ ⊔ v₃)} ⟨ N2 , N3 ⟩ ⊑-dist =
  ⊔-intro N2 N3
sub-ℱ d (⊑-trans x₁ x₂) = sub-ℱ (sub-ℱ d x₂) x₁
```

With this subsumption property in hand, we can prove the forward direction of the semantic equation for lambda. The proof is by induction on the semantics, using `sub-ℱ` in the case for the `sub` rule.

```
ℰƛ→ℱℰ : ∀{Γ}{γ : Env Γ}{N : Γ , ★ ⊢ ★}{v : Value}
  → ℰ (ƛ N) γ v
    ------------
  → ℱ (ℰ N) γ v
ℰƛ→ℱℰ (↦-intro d) = d
ℰƛ→ℱℰ ⊥-intro = tt
```

```
ℰƛ→𝔽ℰ (⊔-intro d₁ d₂) = ⟨ ℰƛ→𝔽ℰ d₁ , ℰƛ→𝔽ℰ d₂ ⟩
ℰƛ→𝔽ℰ (sub d lt) = sub-𝔽 (ℰƛ→𝔽ℰ 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 = ℰƛ→𝔽ℰ{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.

```
𝔽ℰ→ℰƛ : ∀{Γ}{γ : Env Γ}{N : Γ , ★ ⊢ ★}{v : Value}
  → 𝔽 (ℰ N) γ v
    ------------
  → ℰ (ƛ N) γ v
𝔽ℰ→ℰƛ {v = ⊥} d = ⊥-intro
𝔽ℰ→ℰƛ {v = v₁ ↦ v₂} d = ↦-intro d
𝔽ℰ→ℰƛ {v = v₁ ⊔ v₂} ⟨ d1 , d2 ⟩ = ⊔-intro (𝔽ℰ→ℰƛ d1) (𝔽ℰ→ℰƛ 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 `ℰ L ≃ ℰ L′` and `ℰ M ≃ ℰ M′` imply `ℰ (L · M) ≃ ℰ (L′ · M′)`. The `app-equiv` equation reduces this to the question of whether the `●` operator is a congruence.

```
●-cong : ∀{Γ}{D₁ D₁′ D₂ D₂′ : Denotation Γ}
  → D₁ ≃ D₁′ → D₂ ≃ D₂′
  → (D₁ ● D₂) ≃ (D₁′ ● D₂′)
●-cong {Γ} d1 d2 γ v = ⟨ (λ x → ●≃ x d1 d2) ,
                         (λ x → ●≃ x (≃-sym d1) (≃-sym d2)) ⟩
  where
  ●≃ : ∀{γ : Env Γ}{v}{D₁ D₁′ D₂ D₂′ : Denotation Γ}
    → (D₁ ● D₂) γ v → D₁ ≃ D₁′ → D₂ ≃ D₂′
    → (D₁′ ● D₂′) γ v
  ●≃ (inj₁ v⊑⊥) eq₁ eq₂ = inj₁ v⊑⊥
  ●≃ {γ} {w} (inj₂ ⟨ v , ⟨ Dv↦w , Dv ⟩ ⟩) eq₁ eq₂ =
    inj₂ ⟨ v , ⟨ proj₁ (eq₁ γ (v ↦ w)) Dv↦w , proj₁ (eq₂ γ v) Dv ⟩ ⟩
```

Again, both directions of the if-and-only-if are proved via a lemma. This time the lemma is proved by cases on `(D₁ ● D₂) γ v`.

With the congruence of `●`, we can prove that application is a congruence by direct equational reasoning.

```
app-cong : ∀{Γ}{L L′ M M′ : Γ ⊢ ★}
  → ℰ L ≃ ℰ L′
  → ℰ M ≃ ℰ M′
    -------------------------
  → ℰ (L · M) ≃ ℰ (L′ · M′)
app-cong {Γ}{L}{L′}{M}{M′} L≅L′ M≅M′ =
  start
    ℰ (L · M)
  ≃⟨ app-equiv ⟩
    ℰ L ● ℰ M
  ≃⟨ ●-cong L≅L′ M≅M′ ⟩
    ℰ L′ ● ℰ M′
  ≃⟨ ≃-sym app-equiv ⟩
    ℰ (L′ · M′)
  □
```

Compositionality

The *compositionality property* states that surrounding two terms that are denotationally equal in the same context produces two programs that are denotationally equal. To make this precise, we define what we mean by "context" and "surround".

A *context* is a program with one hole in it. The following data definition `Ctx` makes this idea explicit. We index the `Ctx` data type with two contexts for variables: one for the hole and one for terms that result from filling the hole.

```
data Ctx : Context → Context → Set where
  ctx-hole : ∀{Γ} → Ctx Γ Γ
  ctx-lam : ∀{Γ Δ} → Ctx (Γ , ★) (Δ , ★) → Ctx (Γ , ★) Δ
  ctx-app-L : ∀{Γ Δ} → Ctx Γ Δ → Δ ⊢ ★ → Ctx Γ Δ
  ctx-app-R : ∀{Γ Δ} → Δ ⊢ ★ → Ctx Γ Δ → Ctx Γ Δ
```

- The constructor `ctx-hole` represents the hole, and in this case the variable context for the hole is the same as the variable context for the term that results from filling the hole.
- The constructor `ctx-lam` takes a `Ctx` and produces a larger one that adds a lambda abstraction at the top. The variable context of the hole stays the same, whereas we remove one variable from the context of the resulting term because it is bound by this lambda abstraction.
- There are two constructions for application, `ctx-app-L` and `ctx-app-R`. The `ctx-app-L` is for when the hole is inside the left-hand term (the operator) and the later is when the hole is inside the right-hand term (the operand).

The action of surrounding a term with a context is defined by the following `plug` function. It is defined by recursion on the context.

```
plug : ∀{Γ}{Δ} → Ctx Γ Δ → Γ ⊢ ★ → Δ ⊢ ★
plug ctx-hole M = M
plug (ctx-lam C) N = ƛ plug C N
plug (ctx-app-L C N) L = (plug C L) · N
plug (ctx-app-R L C) M = L · (plug C M)
```

We are ready to state and prove the compositionality principle. Given two terms `M` and `N` that are denotationally equal, plugging them both into an arbitrary context `C` produces two programs that are denotationally equal.

```
compositionality : ∀{Γ Δ}{C : Ctx Γ Δ} {M N : Γ ⊢ ★}
  → ℰ M ≃ ℰ N
    ---------------------------
  → ℰ (plug C M) ≃ ℰ (plug C N)
compositionality {C = ctx-hole} M≃N =
  M≃N
compositionality {C = ctx-lam C′} M≃N =
  lam-cong (compositionality {C = C′} M≃N)
compositionality {C = ctx-app-L C′ L} M≃N =
  app-cong (compositionality {C = C′} M≃N) λ γ v → ⟨ (λ x → x) , (λ x → x) ⟩
compositionality {C = ctx-app-R L C′} M≃N =
  app-cong (λ γ v → ⟨ (λ x → x) , (λ x → x) ⟩) (compositionality {C = C′} M≃N)
```

The proof is a straightforward induction on the context `C`, using the congruence properties `lam-cong` and `app-cong` that we established above.

The denotational semantics defined as a function

Having established the three equations `var-equiv`, `lam-equiv`, and `app-equiv`, one should be able to define the denotational semantics as a recursive function over the input term `M`. Indeed, we define the following function `⟦ M ⟧` that maps terms to denotations, using the auxiliary curry `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 —↠ ƛ N  implies  ℰ L ≃ ℰ (ƛ N)
```

The proof is by induction on the reduction sequence, so the main lemma concerns a single reduction step. We prove that if any term `M` steps to a term `N`, then `M` and `N` are denotationally equal. We shall prove each direction of this if-and-only-if separately. One direction will look just like a type preservation proof. The other direction is like proving type preservation for reduction going in reverse. Recall that type preservation is sometimes called subject reduction. Preservation in reverse is a well-known property and is called *subject expansion*. It is also well-known that subject expansion is false for most typed lambda calculi!

Imports

```
open import Relation.Binary.PropositionalEquality
  using (_≡_; _≢_; refl; sym; cong; cong₂; cong-app)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import Agda.Primitive using (lzero)
open import Relation.Nullary using (¬_)
open import Relation.Nullary.Negation using (contradiction)
open import Data.Empty using (⊥-elim)
open import Relation.Nullary using (Dec; yes; no)
open import Function using (_∘_)
open import plfa.part2.Untyped
     using (Context; _,_; _∋_; _⊢_; ★; Z; S_; `_; ƛ_; _·_;
            subst; _[_]; subst-zero; ext; rename; exts;
            _—→_; ξ₁; ξ₂; β; ζ; _—↠_; _—→⟨_⟩_; _∎)
open import plfa.part2.Substitution using (Rename; Subst; ids)
open import plfa.part3.Denotational
     using (Value; ⊥; Env; _⊢_↓_; _`,_; _⊑_; _`⊑_; `⊥; _`⊔_; init; last; init-last;
            ⊑-refl; ⊑-trans; `⊑-refl; ⊑-env; ⊑-env-conj-R1; ⊑-env-conj-R2; up-env;
            var; ↦-elim; ↦-intro; ⊥-intro; ⊔-intro; sub;
            rename-pres; ℰ; _≃_; ≃-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; ℰ; _≃_; _iff_;
            ⊑-trans; ⊑-conj-R1; ⊑-conj-R2; ⊑-conj-L; ⊑-refl; ⊑-fun; ⊑-bot; ⊑-dist;
            sub-inv-fun)
open import plfa.part3.Soundness using (soundness)
```

The property of being greater or equal to a function

We define the following short-hand for saying that a value is greater-than or equal to a function value.

```
above-fun : Value → Set
above-fun u = Σ[ v ∈ Value ] Σ[ w ∈ Value ] v ↦ w ⊑ u
```

If a value `u` is greater than a function, then an even greater value `u'` is too.

```
above-fun-⊑ : ∀{u u' : Value}
  → above-fun u → u ⊑ u'
    -------------------
  → above-fun u'
above-fun-⊑ ⟨ v , ⟨ w , lt' ⟩ ⟩ lt = ⟨ v , ⟨ w , ⊑-trans lt' lt ⟩ ⟩
```

The bottom value `⊥` is not greater than a function.

```
above-fun⊥ : ¬ above-fun ⊥
above-fun⊥ ⟨ v , ⟨ w , lt ⟩ ⟩
    with sub-inv-fun lt
... | ⟨ Γ , ⟨ f , ⟨ Γ⊆⊥ , ⟨ lt1 , lt2 ⟩ ⟩ ⟩ ⟩
    with all-funs∈ f
... | ⟨ A , ⟨ B , m ⟩ ⟩
    with Γ⊆⊥ m
... | ()
```

If the join of two values `u` and `u'` is greater than a function, then at least one of them is too.

```
above-fun-⊔ : ∀{u u'}
  → above-fun (u ⊔ u')
  → above-fun u ⊎ above-fun u'
above-fun-⊔{u}{u'} ⟨ v , ⟨ w , v↦w⊑u⊔u' ⟩ ⟩
    with sub-inv-fun v↦w⊑u⊔u'
... | ⟨ Γ , ⟨ f , ⟨ Γ⊆u⊔u' , ⟨ lt1 , lt2 ⟩ ⟩ ⟩ ⟩
    with all-funs∈ f
... | ⟨ A , ⟨ B , m ⟩ ⟩
    with Γ⊆u⊔u' m
... | inj₁ x = inj₁ ⟨ A , ⟨ B , (∈→⊑ x) ⟩ ⟩
... | inj₂ x = inj₂ ⟨ A , ⟨ B , (∈→⊑ x) ⟩ ⟩
```

On the other hand, if neither of `u` and `u'` is greater than a function, then their join is also not greater than a function.

```
not-above-fun-⊔ : ∀{u u' : Value}
  → ¬ above-fun u → ¬ above-fun u'
  → ¬ above-fun (u ⊔ u')
not-above-fun-⊔ naf1 naf2 af12
  with above-fun-⊔ af12
... | inj₁ af1 = contradiction af1 naf1
... | inj₂ af2 = contradiction af2 naf2
```

The converse is also true. If the join of two values is not above a function, then neither of them is individually.

```
not-above-fun-⊔-inv : ∀{u u' : Value} → ¬ above-fun (u ⊔ u')
  → ¬ above-fun u × ¬ above-fun u'
not-above-fun-⊔-inv af = ⟨ f af , g af ⟩
  where
    f : ∀{u u' : Value} → ¬ above-fun (u ⊔ u') → ¬ above-fun u
    f{u}{u'} af12 ⟨ v , ⟨ w , lt ⟩ ⟩ =
      contradiction ⟨ v , ⟨ w , ⊑-conj-R1 lt ⟩ ⟩ af12
    g : ∀{u u' : Value} → ¬ above-fun (u ⊔ u') → ¬ above-fun u'
    g{u}{u'} af12 ⟨ v , ⟨ w , lt ⟩ ⟩ =
      contradiction ⟨ v , ⟨ w , ⊑-conj-R2 lt ⟩ ⟩ af12
```

The property of being greater than a function value is decidable, as exhibited by the following function.

```
above-fun? : (v : Value) → Dec (above-fun v)
above-fun? ⊥ = no above-fun⊥
above-fun? (v ↦ w) = yes ⟨ v , ⟨ w , ⊑-refl ⟩ ⟩
above-fun? (u ⊔ u')
  with above-fun? u | above-fun? u'
... | yes ⟨ v , ⟨ w , lt ⟩ ⟩ | _ = yes ⟨ v , ⟨ w , (⊑-conj-R1 lt) ⟩ ⟩
... | no _ | yes ⟨ v , ⟨ w , lt ⟩ ⟩ = yes ⟨ v , ⟨ w , (⊑-conj-R2 lt) ⟩ ⟩
... | no x | no y = no (not-above-fun-⊔ x y)
```

Relating values to closures

Next we relate semantic values to closures. The relation `𝕍` is for closures whose term is a lambda abstraction, i.e., in weak-head normal form (WHNF). The relation 𝔼 is for any closure. Roughly speaking, `𝔼 v c` will hold if, when `v` is greater than a function value, `c` evaluates to a closure `c'` in WHNF and `𝕍 v c'`. Regarding `𝕍 v c`, it will hold when `c` is in WHNF, and if `v` is a function, the body of `c` evaluates according to `v`.

```
𝕍 : Value → Clos → Set
𝔼 : Value → Clos → Set
```

We define `𝕍` as a function from values and closures to `Set` and not as a data type because it is mutually recursive with `𝔼` in a negative position (to the left of an implication). We first perform case analysis on the term in the closure. If the term is a variable or application, then `𝕍` is false (`Bot`). If the term is a lambda abstraction, we define `𝕍` by recursion on the value, which we describe below.

```
𝕍 v (clos (` x₁) γ) = Bot
𝕍 v (clos (M · M₁) γ) = Bot
𝕍 ⊥ (clos (ƛ M) γ) = ⊤
𝕍 (v ↦ w) (clos (ƛ N) γ) =
    (∀{c : Clos} → 𝔼 v c → above-fun w → Σ[ c' ∈ Clos ]
      (γ ,' c) ⊢ N ⇓ c' × 𝕍 w c')
𝕍 (u ⊔ v) (clos (ƛ N) γ) = 𝕍 u (clos (ƛ N) γ) × 𝕍 v (clos (ƛ N) γ)
```

- If the value is `⊥`, then the result is true (`⊤`).
- If the value is a join (u ⊔ v), then the result is the pair (conjunction) of 𝕍 is true for both u and v.
- The important case is for a function value `v ↦ w` and closure `clos (ƛ N) γ`. Given any closure `c` such that `𝔼 v c`, if `w` is greater than a function, then `N` evaluates (with `γ` extended with `c`) to some closure `c'` and we have `𝕍 w c'`.

The definition of `𝔼` is straightforward. If `v` is a greater than a function, then `M` evaluates to a closure related to `v`.

```
𝔼 v (clos M γ') = above-fun v → Σ[ c ∈ Clos ] γ' ⊢ M ⇓ c × 𝕍 v c
```

The proof of the main lemma is by induction on `γ ⊢ M ↓ v`, so it goes underneath lambda abstractions and must therefore reason about open terms (terms with variables). So we must relate environments of semantic values to environments of closures. In the following, `𝔾` relates `γ` to `γ'` if the corresponding values and closures are related by `𝔼`.

```
𝔾 : ∀{Γ} → Env Γ → ClosEnv Γ → Set
𝔾 {Γ} γ γ' = ∀{x : Γ ∋ ★} → 𝔼 (γ x) (γ' x)

𝔾-∅ : 𝔾 `∅ ∅'
𝔾-∅ {()}

𝔾-ext : ∀{Γ}{γ : Env Γ}{γ' : ClosEnv Γ}{v c}
      → 𝔾 γ γ' → 𝔼 v c → 𝔾 (γ `, v) (γ' ,' c)
𝔾-ext {Γ} {γ} {γ'} g e {Z} = e
𝔾-ext {Γ} {γ} {γ'} g e {S x} = g
```

We need a few properties of the `𝕍` and `𝔼` relations. The first is that a closure in the `𝕍` relation must be in weak-head normal form. We define WHNF has follows.

```
data WHNF : ∀ {Γ A} → Γ ⊢ A → Set where
  ƛ_ : ∀ {Γ} {N : Γ , ★ ⊢ ★}
     → WHNF (ƛ N)
```

The proof goes by cases on the term in the closure.

```
𝕍→WHNF : ∀{Γ}{γ : ClosEnv Γ}{M : Γ ⊢ ★}{v}
       → 𝕍 v (clos M γ) → WHNF M
𝕍→WHNF {M = ` x} {v} ()
𝕍→WHNF {M = ƛ N} {v} vc = ƛ_
𝕍→WHNF {M = L · M} {v} ()
```

Next we have an introduction rule for `𝕍` that mimics the `⊔-intro` rule. If both `u` and `v` are related to a closure `c`, then their join is too.

```
𝕍⊔-intro : ∀{c u v}
         → 𝕍 u c → 𝕍 v c
           ---------------
         → 𝕍 (u ⊔ v) c
𝕍⊔-intro {clos (` x) γ} () vc
𝕍⊔-intro {clos (ƛ N) γ} uc vc = ⟨ uc , vc ⟩
𝕍⊔-intro {clos (L · M) γ} () vc
```

In a moment we prove that `𝕍` is preserved when going from a greater value to a lesser value: if `𝕍 v c` and `v' ⊑ v`, then `𝕍 v' c`. This property, named `𝕍-sub`, is needed by the main lemma in the case for the `sub` rule.

To prove `𝕍-sub`, we in turn need the following property concerning values that are not greater than a function, that is, values that are equivalent to `⊥`. In such cases, `𝕍 v (clos (ƛ N) γ')` is trivially true.

```
not-above-fun-𝕍 : ∀{v : Value}{Γ}{γ' : ClosEnv Γ}{N : Γ , ★ ⊢ ★ }
  → ¬ above-fun v
    -------------------
  → 𝕍 v (clos (ƛ N) γ')
not-above-fun-𝕍 {⊥} af = tt
not-above-fun-𝕍 {v ↦ v'} af = ⊥-elim (contradiction ⟨ v , ⟨ v' , ⊑-refl ⟩ ⟩ af)
not-above-fun-𝕍 {v₁ ⊔ v₂} af
  with not-above-fun-⊔-inv af
```

```
... | ⟨ af1 , af2 ⟩ = ⟨ not-above-fun-𝕍 af1 , not-above-fun-𝕍 af2 ⟩
```

The proofs of `𝕍-sub` and `𝔼-sub` are intertwined.

```
sub-𝕍 : ∀{c : Clos}{v v'} → 𝕍 v c → v' ⊑ v → 𝕍 v' c
sub-𝔼 : ∀{c : Clos}{v v'} → 𝔼 v c → v' ⊑ v → 𝔼 v' c
```

We prove `𝕍-sub` by case analysis on the closure's term, to dispatch the cases for variables and application. We then proceed by induction on `v' ⊑ v`. We describe each case below.

```
sub-𝕍 {clos (` x) γ} {v} () lt
sub-𝕍 {clos (L · M) γ} () lt
sub-𝕍 {clos (ƛ N) γ} vc ⊑-bot = tt
sub-𝕍 {clos (ƛ N) γ} vc (⊑-conj-L lt1 lt2) = ⟨ (sub-𝕍 vc lt1) , sub-𝕍 vc lt2 ⟩
sub-𝕍 {clos (ƛ N) γ} ⟨ vv1 , vv2 ⟩ (⊑-conj-R1 lt) = sub-𝕍 vv1 lt
sub-𝕍 {clos (ƛ N) γ} ⟨ vv1 , vv2 ⟩ (⊑-conj-R2 lt) = sub-𝕍 vv2 lt
sub-𝕍 {clos (ƛ N) γ} vc (⊑-trans{v = v₂} lt1 lt2) = sub-𝕍 (sub-𝕍 vc lt2) lt1
sub-𝕍 {clos (ƛ N) γ} vc (⊑-fun lt1 lt2) ev1 sf
    with vc (sub-𝔼 ev1 lt1) (above-fun-⊑ sf lt2)
... | ⟨ c , ⟨ Nc , v4 ⟩ ⟩ = ⟨ c , ⟨ Nc , sub-𝕍 v4 lt2 ⟩ ⟩
sub-𝕍 {clos (ƛ N) γ} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c sf
    with above-fun? w | above-fun? w'
... | yes af2 | yes af3
    with vcw ev1c af2 | vcw' ev1c af3
... | ⟨ clos L δ , ⟨ L⇓c₂ , 𝕍w ⟩ ⟩
    | ⟨ c₃ , ⟨ L⇓c₃ , 𝕍w' ⟩ ⟩ rewrite ⇓-determ L⇓c₃ L⇓c₂ with 𝕍→WHNF 𝕍w
... | ƛ_ =
      ⟨ clos L δ , ⟨ L⇓c₂ , ⟨ 𝕍w , 𝕍w' ⟩ ⟩ ⟩
sub-𝕍 {c} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c sf
    | yes af2 | no naf3
    with vcw ev1c af2
... | ⟨ clos {Γ'} L γ₁ , ⟨ L⇓c2 , 𝕍w ⟩ ⟩
    with 𝕍→WHNF 𝕍w
... | ƛ_ {N = N'} =
      let 𝕍w' = not-above-fun-𝕍{w'}{Γ'}{γ₁}{N'} naf3 in
      ⟨ clos (ƛ N') γ₁ , ⟨ L⇓c2 , 𝕍⊔-intro 𝕍w 𝕍w' ⟩ ⟩
sub-𝕍 {c} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c sf
    | no naf2 | yes af3
    with vcw' ev1c af3
... | ⟨ clos {Γ'} L γ₁ , ⟨ L⇓c3 , 𝕍w'c ⟩ ⟩
    with 𝕍→WHNF 𝕍w'c
... | ƛ_ {N = N'} =
      let 𝕍wc = not-above-fun-𝕍{w}{Γ'}{γ₁}{N'} naf2 in
      ⟨ clos (ƛ N') γ₁ , ⟨ L⇓c3 , 𝕍⊔-intro 𝕍wc 𝕍w'c ⟩ ⟩
sub-𝕍 {c} {v ↦ w ⊔ v ↦ w'} ⟨ vcw , vcw' ⟩ ⊑-dist ev1c ⟨ v' , ⟨ w'' , lt ⟩ ⟩
    | no naf2 | no naf3
    with above-fun-⊔ ⟨ v' , ⟨ w'' , lt ⟩ ⟩
... | inj₁ af2 = ⊥-elim (contradiction af2 naf2)
... | inj₂ af3 = ⊥-elim (contradiction af3 naf3)
```

- Case `⊑-bot`. We immediately have `𝕍 ⊥ (clos (ƛ N) γ)`.
- Case `⊑-conj-L`.

```
    v₁' ⊑ v     v₂' ⊑ v
    -------------------
```

```
(v₁' ⊔ v₂') ⊑ v
```

The induction hypotheses gives us `𝕍 v₁' (clos (ƛ N) γ)` and `𝕍 v₂' (clos (ƛ N) γ)`, which is all we need for this case.

- Case `⊑-conj-R1`.

  ```
  v' ⊑ v₁
  -------------
  v' ⊑ (v₁ ⊔ v₂)
  ```

  The induction hypothesis gives us `𝕍 v' (clos (ƛ N) γ)`.

- Case `⊑-conj-R2`.

  ```
  v' ⊑ v₂
  -------------
  v' ⊑ (v₁ ⊔ v₂)
  ```

  Again, the induction hypothesis gives us `𝕍 v' (clos (ƛ N) γ)`.

- Case `⊑-trans`.

  ```
  v' ⊑ v₂   v₂ ⊑ v
  -----------------
       v' ⊑ v
  ```

  The induction hypothesis for `v₂ ⊑ v` gives us `𝕍 v₂ (clos (ƛ N) γ)`. We apply the induction hypothesis for `v' ⊑ v₂` to conclude that `𝕍 v' (clos (ƛ N) γ)`.

- Case `⊑-dist`. This case is the most difficult. We have

  ```
  𝕍 (v ↦ w) (clos (ƛ N) γ)
  𝕍 (v ↦ w') (clos (ƛ N) γ)
  ```

  and need to show that

  ```
  𝕍 (v ↦ (w ⊔ w')) (clos (ƛ N) γ)
  ```

  Let `c` be an arbitrary closure such that `𝔼 v c`. Assume `w ⊔ w'` is greater than a function. Unfortunately, this does not mean that both `w` and `w'` are above functions. But thanks to the lemma `above-fun-⊔`, we know that at least one of them is greater than a function.

  - Suppose both of them are greater than a function. Then we have `γ ⊢ N ⇓ clos L δ` and `𝕍 w (clos L δ)`. We also have `γ ⊢ N ⇓ c₃` and `𝕍 w' c₃`. Because the big-step semantics is deterministic, we have `c₃ ≡ clos L δ`. Also, from `𝕍 w (clos L δ)` we know that `L ≡ ƛ N'` for some `N'`. We conclude that `𝕍 (w ⊔ w') (clos (ƛ N') δ)`.
  - Suppose one of them is greater than a function and the other is not: say `above-fun w` and `¬ above-fun w'`. Then from `𝕍 (v ↦ w) (clos (ƛ N) γ)` we have `γ ⊢ N ⇓ clos L γ₁` and `𝕍 w (clos L γ₁)`. From this we have `L ≡ ƛ N'` for some `N'`. Meanwhile, from `¬ above-fun w'` we have `𝕍 w' (clos L γ₁)`. We conclude that `𝕍 (w ⊔ w') (clos (ƛ N') γ₁)`.

The proof of `sub-𝔼` is direct and explained below.

```
sub-𝔼 {clos M γ} {v} {v'} 𝔼v v'⊑v fv'
  with 𝔼v (above-fun-⊑ fv' v'⊑v)
... | ⟨ c , ⟨ M⇓c , 𝕍v ⟩ ⟩ =
      ⟨ c , ⟨ M⇓c , sub-𝕍 𝕍v v'⊑v ⟩ ⟩
```

From `above-fun v'` and `v' ⊑ v` we have `above-fun v`. Then with `𝔼 v c` we obtain a closure `c` such that `γ ⊢ M ⇓ c` and `𝕍 v c`. We conclude with an application of `sub-𝕍` with `v' ⊑ v` to show `𝕍 v' c`.

Programs with function denotation terminate via call-by-name

The main lemma proves that if a term has a denotation that is above a function, then it terminates via call-by-name. More formally, if `γ ⊢ M ↓ v` and `𝔾 γ γ'`, then `𝔼 v (clos M γ')`. The proof is by induction on the derivation of `γ ⊢ M ↓ v` we discuss each case below.

The following lemma, kth-x, is used in the case for the `var` rule.

```
kth-x : ∀{Γ}{γ' : ClosEnv Γ}{x : Γ ∋ ★}
  → Σ[ Δ ∈ Context ] Σ[ δ ∈ ClosEnv Δ ] Σ[ M ∈ Δ ⊢ ★ ]
            γ' x ≡ clos M δ
kth-x{γ' = γ'}{x = x} with γ' x
... | clos{Γ = Δ} M δ = ⟨ Δ , ⟨ δ , ⟨ M , refl ⟩ ⟩ ⟩
```

```
↓→𝔼 : ∀{Γ}{γ : Env Γ}{γ' : ClosEnv Γ}{M : Γ ⊢ ★ }{v}
            → 𝔾 γ γ' → γ ⊢ M ↓ v → 𝔼 v (clos M γ')
↓→𝔼 {Γ} {γ} {γ'} 𝔾γγ' (var{x = x}) fγx
    with kth-x{Γ}{γ'}{x} | 𝔾γγ'{x = x}
... | ⟨ Δ , ⟨ δ , ⟨ M' , eq ⟩ ⟩ ⟩ | 𝔾γγ'x rewrite eq
    with 𝔾γγ'x fγx
... | ⟨ c , ⟨ M'⇓c , 𝕍γx ⟩ ⟩ =
      ⟨ c , ⟨ (⇓-var eq M'⇓c) , 𝕍γx ⟩ ⟩
↓→𝔼 {Γ} {γ} {γ'} 𝔾γγ' (↦-elim{L = L}{M = M}{v = v₁}{w = v} d₁ d₂) fv
    with ↓→𝔼 𝔾γγ' d₁ ⟨ v₁ , ⟨ v , ⊑-refl ⟩ ⟩
... | ⟨ clos L' δ , ⟨ L⇓L' , 𝕍v₁↦v ⟩ ⟩
    with 𝕍→WHNF 𝕍v₁↦v
... | ƛ_ {N = N}
    with 𝕍v₁↦v {clos M γ'} (↓→𝔼 𝔾γγ' d₂) fv
... | ⟨ c' , ⟨ N⇓c' , 𝕍v ⟩ ⟩ =
    ⟨ c' , ⟨ ⇓-app L⇓L' N⇓c' , 𝕍v ⟩ ⟩
↓→𝔼 {Γ} {γ} {γ'} 𝔾γγ' (↦-intro{N = N}{v = v}{w = w} d) fv↦w =
    ⟨ clos (ƛ N) γ' , ⟨ ⇓-lam , E ⟩ ⟩
    where E : {c : Clos} → 𝔼 v c → above-fun w
              → Σ[ c' ∈ Clos ] (γ' ,' c) ⊢ N ⇓ c' × 𝕍 w c'
          E {c} 𝔼vc fw = ↓→𝔼 (λ {x} → 𝔾-ext{Γ}{γ}{γ'} 𝔾γγ' 𝔼vc {x}) d fw
↓→𝔼 𝔾γγ' ⊥-intro f⊥ = ⊥-elim (above-fun⊥ f⊥)
↓→𝔼 𝔾γγ' (⊔-intro{v = v₁}{w = v₂} d₁ d₂) fv12
    with above-fun? v₁ | above-fun? v₂
... | yes fv1 | yes fv2
    with ↓→𝔼 𝔾γγ' d₁ fv1 | ↓→𝔼 𝔾γγ' d₂ fv2
... | ⟨ c₁ , ⟨ M⇓c₁ , 𝕍v₁ ⟩ ⟩ | ⟨ c₂ , ⟨ M⇓c₂ , 𝕍v₂ ⟩ ⟩
    rewrite ⇓-determ M⇓c₂ M⇓c₁ =
    ⟨ c₁ , ⟨ M⇓c₁ , 𝕍⊔-intro 𝕍v₁ 𝕍v₂ ⟩ ⟩
```

```
↓→𝔼 𝔾γγ' (⊔-intro{v = v₁}{w = v₂} d₁ d₂) fv12 | yes fv1 | no nfv2
    with ↓→𝔼 𝔾γγ' d₁ fv1
... | ⟨ clos {Γ'} M' γ₁ , ⟨ M⇓c₁ , 𝕍v₁ ⟩ ⟩
    with 𝕍→WHNF 𝕍v₁
... | ƛ_ {N = N} =
    let 𝕍v₂ = not-above-fun-𝕍{v₂}{Γ'}{γ₁}{N} nfv2 in
    ⟨ clos (ƛ N) γ₁ , ⟨ M⇓c₁ , 𝕍⊔-intro 𝕍v₁ 𝕍v₂ ⟩ ⟩
↓→𝔼 𝔾γγ' (⊔-intro{v = v₁}{w = v₂} d₁ d₂) fv12 | no nfv1 | yes fv2
    with ↓→𝔼 𝔾γγ' d₂ fv2
... | ⟨ clos {Γ'} M' γ₁ , ⟨ M'⇓c₂ , 𝕍2c ⟩ ⟩
    with 𝕍→WHNF 𝕍2c
... | ƛ_ {N = N} =
    let 𝕍1c = not-above-fun-𝕍{v₁}{Γ'}{γ₁}{N} nfv1 in
    ⟨ clos (ƛ N) γ₁ , ⟨ M'⇓c₂ , 𝕍⊔-intro 𝕍1c 𝕍2c ⟩ ⟩
↓→𝔼 𝔾γγ' (⊔-intro d₁ d₂) fv12 | no nfv1 | no nfv2
    with above-fun-⊔ fv12
... | inj₁ fv1 = ⊥-elim (contradiction fv1 nfv1)
... | inj₂ fv2 = ⊥-elim (contradiction fv2 nfv2)
↓→𝔼 {Γ} {γ} {γ'} {M} {v'} 𝔾γγ' (sub{v = v} d v'⊑v) fv'
    with ↓→𝔼 {Γ} {γ} {γ'} {M} 𝔾γγ' d (above-fun-⊑ fv' v'⊑v)
... | ⟨ c , ⟨ M⇓c , 𝕍v ⟩ ⟩ =
      ⟨ c , ⟨ M⇓c , sub-𝕍 𝕍v v'⊑v ⟩ ⟩
```

- Case `var`. Looking up `x` in `γ'` yields some closure, `clos M' δ`, and from `𝔾 γ γ'` we have `𝔼 (γ x) (clos M' δ)`. With the premise `above-fun (γ x)`, we obtain a closure `c` such that `δ ⊢ M' ⇓ c` and `𝕍 (γ x) c`. To conclude `γ' ⊢ x ⇓ c` via `⇓-var`, we need `γ' x ≡ clos M' δ`, which is obvious, but it requires some Agda shananigans via the `kth-x` lemma to get our hands on it.

- Case `↦-elim`. We have `γ ⊢ L · M ↓ v`. The induction hypothesis for `γ ⊢ L ↓ v₁ ↦ v` gives us `γ' ⊢ L ⇓ clos L' δ` and `𝕍 v (clos L' δ)`. Of course, `L' ≡ ƛ N` for some `N`. By the induction hypothesis for `γ ⊢ M ↓ v₁`, we have `𝔼 v₁ (clos M γ')`. Together with the premise `above-fun v` and `𝕍 v (clos L' δ)`, we obtain a closure `c'` such that `δ ⊢ N ⇓ c'` and `𝕍 v c'`. We conclude that `γ' ⊢ L · M ⇓ c'` by rule `⇓-app`.

- Case `↦-intro`. We have `γ ⊢ ƛ N ↓ v ↦ w`. We immediately have `γ' ⊢ ƛ M ⇓ clos (ƛ M) γ'` by rule `⇓-lam`. But we also need to prove `𝕍 (v ↦ w) (clos (ƛ N) γ')`. Let `c` by an arbitrary closure such that `𝔼 v c`. Suppose `v'` is greater than a function value. We need to show that `γ' , c ⊢ N ⇓ c'` and `𝕍 v' c'` for some `c'`. We prove this by the induction hypothesis for `γ , v ⊢ N ↓ v'` but we must first show that `𝔾 (γ , v) (γ' , c)`. We prove that by the lemma `𝔾-ext`, using facts `𝔾 γ γ'` and `𝔼 v c`.

- Case `⊥-intro`. We have the premise `above-fun ⊥`, but that’s impossible.

- Case `⊔-intro`. We have `γ ⊢ M ↓ (v₁ ⊔ v₂)` and `above-fun (v₁ ⊔ v₂)` and need to show `γ' ⊢ M ↓ c` and `𝕍 (v₁ ⊔ v₂) c` for some `c`. Again, by `above-fun-⊔`, at least one of `v₁` or `v₂` is greater than a function.

  - Suppose both `v₁` and `v₂` are greater than a function value. By the induction hypotheses for `γ ⊢ M ↓ v₁` and `γ ⊢ M ↓ v₂` we have `γ' ⊢ M ⇓ c₁`, `𝕍 v₁ c₁`, `γ' ⊢ M ⇓ c₂`, and `𝕍 v₂ c₂` for some `c₁` and `c₂`. Because `⇓` is deterministic, we have `c₂ ≡ c₁`. Then by `𝕍⊔-intro` we conclude that `𝕍 (v₁ ⊔ v₂) c₁`.

  - Without loss of generality, suppose `v₁` is greater than a function value but `v₂` is not. By the induction hypotheses for `γ ⊢ M ↓ v₁`, and using `𝕍→WHNF`, we have `γ' ⊢ M ⇓ clos (ƛ N) γ₁` and `𝕍 v₁ (clos (ƛ N) γ₁)`. Then because `v₂` is not greater than a function, we also have `𝕍 v₂ (clos (ƛ N) γ₁)`. We conclude that `𝕍 (v₁ ⊔ v₂) (clos (ƛ N) γ₁)`.

- Case `sub`. We have `γ ⊢ M ↓ v`, `v' ⊑ v`, and `above-fun v'`. We need to show that `γ' ⊢ M ⇓ c` and `𝕍 v' c` for some `c`. We have `above-fun v` by `above-fun-⊑`, so the induction hypothesis for `γ ⊢ M ↓ v` gives us a closure `c` such that `γ' ⊢ M ⇓ c` and `𝕍 v c`. We conclude that `𝕍 v' c` by `sub-𝕍`.

Proof of denotational adequacy

From the main lemma we can directly show that `ℰ M ≃ ℰ (ƛ N)` implies that `M` big-steps to a lambda, i.e., `∅ ⊢ M ⇓ clos (ƛ N′) γ`.

```
↓→⇓ : ∀{M : ∅ ⊢ ★}{N : ∅ , ★ ⊢ ★} → ℰ M ≃ ℰ (ƛ N)
       → Σ[ Γ ∈ Context ] Σ[ N′ ∈ (Γ , ★ ⊢ ★) ] Σ[ γ ∈ ClosEnv Γ ]
         ∅' ⊢ M ⇓ clos (ƛ N′) γ
↓→⇓{M}{N} eq
    with ↓→𝔼 𝔾-∅ ((proj₂ (eq `∅ (⊥ ↦ ⊥))) (↦-intro ⊥-intro))
                 ⟨ ⊥ , ⟨ ⊥ , ⊑-refl ⟩ ⟩
... | ⟨ clos {Γ} M′ γ , ⟨ M⇓c , Vc ⟩ ⟩
    with 𝕍→WHNF Vc
... | ƛ_ {N = N′} =
    ⟨ Γ , ⟨ N′ , ⟨ γ , M⇓c ⟩ ⟩ ⟩
```

The proof goes as follows. We derive `∅ ⊢ ƛ N ↓ ⊥ ↦ ⊥` and then `ℰ M ≃ ℰ (ƛ N)` gives us `∅ ⊢ M ↓ ⊥ ↦ ⊥`. We conclude by applying the main lemma to obtain `∅ ⊢ M ⇓ clos (ƛ N′) γ` for some `N′` and `γ`.

Now to prove the adequacy property. We apply the above lemma to obtain `∅ ⊢ M ⇓ clos (ƛ N′) γ` and then apply `cbn→reduce` to conclude.

```
adequacy : ∀{M : ∅ ⊢ ★}{N : ∅ , ★ ⊢ ★}
  → ℰ M ≃ ℰ (ƛ N)
  → Σ[ N′ ∈ (∅ , ★ ⊢ ★) ]
    (M —↠ ƛ N′)
adequacy{M}{N} eq
    with ↓→⇓ eq
... | ⟨ Γ , ⟨ N′ , ⟨ γ , M⇓ ⟩ ⟩ ⟩ =
    cbn→reduce M⇓
```

Call-by-name is equivalent to beta reduction

As promised, we return to the question of whether call-by-name evaluation is equivalent to beta reduction. In chapter BigStep we established the forward direction: that if call-by-name produces a result, then the program beta reduces to a lambda abstraction (`cbn→reduce`). We now prove the backward direction of the if-and-only-if, leveraging our results about the denotational semantics.

```
reduce→cbn : ∀ {M : ∅ ⊢ ★} {N : ∅ , ★ ⊢ ★}
           → M —↠ ƛ N
           → Σ[ Δ ∈ Context ] Σ[ N′ ∈ Δ , ★ ⊢ ★ ] Σ[ δ ∈ ClosEnv Δ ]
             ∅' ⊢ M ⇓ clos (ƛ N′) δ
reduce→cbn M—↠ƛN = ↓→⇓ (soundness M—↠ƛN)
```

Suppose `M —↠ ƛ N`. Soundness of the denotational semantics gives us `ℰ M ≃ ℰ (ƛ N)`. Then by `↓→⇓` we conclude that `∅' ⊢ M ⇓ clos (ƛ N′) δ` for some `N′` and `δ`.

Putting the two directions of the if-and-only-if together, we establish that call-by-name evaluation is equivalent to beta reduction in the following sense.

```
cbn↔reduce : ∀ {M : ∅ ⊢ ★}
           → (Σ[ N ∈ ∅ , ★ ⊢ ★ ] (M —↠ ƛ N))
             iff
             (Σ[ Δ ∈ Context ] Σ[ N′ ∈ Δ , ★ ⊢ ★ ] Σ[ δ ∈ ClosEnv Δ ]
               ∅' ⊢ M ⇓ clos (ƛ N′) δ)
cbn↔reduce {M} = ⟨ (λ x → reduce→cbn (proj₂ x)) ,
                   (λ x → cbn→reduce (proj₂ (proj₂ (proj₂ x)))) ⟩
```

Unicode

This chapter uses the following unicode:

```
𝔼  U+1D53C  MATHEMATICAL DOUBLE-STRUCK CAPITAL E (\bE)
𝔾  U+1D53E  MATHEMATICAL DOUBLE-STRUCK CAPITAL G (\bG)
𝕍  U+1D53E  MATHEMATICAL DOUBLE-STRUCK CAPITAL V (\bV)
```

Chapter 24

ContextualEquivalence: Denotational equality implies contextual equivalence

```
module plfa.part3.ContextualEquivalence where
```

Imports

```
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
    renaming (_,_ to ⟨_,_⟩)
open import plfa.part2.Untyped using (_⊢_; ★; ∅; _,_; ƛ_; _—↠_)
open import plfa.part2.BigStep using (_⊢_⇓_; cbn→reduce)
open import plfa.part3.Denotational using (ℰ; _≃_; ≃-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

[illegible]

Appendix A

Substitution: Substitution in the untyped lambda calculus

```
module plfa.part2.Substitution where
```

Introduction

The primary purpose of this chapter is to prove that substitution commutes with itself. Barendregt (1984) refers to this as the substitution lemma:

```
M [x:=N] [y:=L] = M [y:=L] [x:= N[y:=L] ]
```

In our setting, with de Bruijn indices for variables, the statement of the lemma becomes:

```
M [ N ] [ L ] ≡  M〔 L 〕[ N [ L ] ]                     (substitution)
```

where the notation `M 〔 L 〕` is for substituting L for index 1 inside M. In addition, because we define substitution in terms of parallel substitution, we have the following generalization, replacing the substitution of L with an arbitrary parallel substitution σ.

```
subst σ (M [ N ]) ≡ (subst (exts σ) M) [ subst σ N ]    (subst-commute)
```

The special case for renamings is also useful.

```
rename ρ (M [ N ]) ≡ (rename (ext ρ) M) [ rename ρ N ]
                                                 (rename-subst-commute)
```

The secondary purpose of this chapter is to define the σ algebra of parallel substitution due to Abadi, Cardelli, Curien, and Levy (1991). The equations of this algebra not only help us prove the substitution lemma, but they are generally useful. Furthermore, when the equations are applied from left to right, they form a rewrite system that *decides* whether any two substitutions are equal.

Imports

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; cong; cong₂; cong-app)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _∎)
open import Function using (_∘_)
open import plfa.part2.Untyped
    using (Type; Context; _⊢_; ★; _∋_; ∅; _,_; Z; S_; `_; ƛ_; _·_;
           rename; subst; ext; exts; _[_]; subst-zero)
```

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

Notation

We introduce the following shorthand for the type of a *renaming* from variables in context `Γ` to variables in context `Δ`.

```
Rename : Context → Context → Set
Rename Γ Δ = ∀{A} → Γ ∋ A → Δ ∋ A
```

Similarly, we introduce the following shorthand for the type of a *substitution* from variables in context `Γ` to terms in context `Δ`.

```
Subst : Context → Context → Set
Subst Γ Δ = ∀{A} → Γ ∋ A → Δ ⊢ A
```

We use the following more succinct notation the `subst` function.

```
⟪_⟫ : ∀{Γ Δ A} → Subst Γ Δ → Γ ⊢ A → Δ ⊢ A
⟪ σ ⟫ = λ M → subst σ M
```

The σ algebra of substitution

Substitutions map de Bruijn indices (natural numbers) to terms, so we can view a substitution simply as a sequence of terms, or more precisely, as an infinite sequence of terms. The σ algebra consists of four operations for building such sequences: identity `ids`, shift `↑`, cons `M • σ`, and sequencing `σ ⨟ τ`. The sequence `0, 1, 2, ...` is constructed by the identity substitution.

```
ids : ∀{Γ} → Subst Γ Γ
ids x = ` x
```

The shift operation `↑` constructs the sequence

```
1, 2, 3, ...
```

and is defined as follows.

```
↑ : ∀{Γ A} → Subst Γ (Γ , A)
↑ x = ` (S x)
```

Given a term `M` and substitution `σ`, the operation `M • σ` constructs the sequence

```
M , σ 0, σ 1, σ 2, ...
```

This operation is analogous to the `cons` operation of Lisp.

```
infixr 6 _•_

_•_ : ∀{Γ Δ A} → (Δ ⊢ A) → Subst Γ Δ → Subst (Γ , A) Δ
(M • σ) Z = M
(M • σ) (S x) = σ x
```

Given two substitutions `σ` and `τ`, the sequencing operation `σ ⨟ τ` produces the sequence

```
⟪τ⟫(σ 0), ⟪τ⟫(σ 1), ⟪τ⟫(σ 2), ...
```

That is, it composes the two substitutions by first applying `σ` and then applying `τ`.

```
infixr 5 _⨟_

_⨟_ : ∀{Γ Δ Σ} → Subst Γ Δ → Subst Δ Σ → Subst Γ Σ
σ ⨟ τ = ⟪ τ ⟫ ∘ σ
```

For the sequencing operation, Abadi et al. use the notation of function composition, writing `σ ∘ τ`, but still with `σ` applied before `τ`, which is the opposite of standard mathematical practice. We instead write `σ ⨟ τ`, because semicolon is the standard notation for forward function composition.

The σ algebra equations

The σ algebra includes the following equations.

```
(sub-head)  ⟪ M • σ ⟫ (` Z) ≡ M
(sub-tail)  ↑ ⨟ (M • σ)    ≡ σ
(sub-η)     (⟪ σ ⟫ (` Z)) • (↑ ⨟ σ) ≡ σ
(Z-shift)   (` Z) • ↑      ≡ ids

(sub-id)    ⟪ ids ⟫ M      ≡ M
(sub-app)   ⟪ σ ⟫ (L · M)  ≡ (⟪ σ ⟫ L) · (⟪ σ ⟫ M)
(sub-abs)   ⟪ σ ⟫ (ƛ N)    ≡ ƛ ⟪ σ ⟫ N
(sub-sub)   ⟪ τ ⟫ ⟪ σ ⟫ M  ≡ ⟪ σ ⨟ τ ⟫ M

(sub-idL)   ids ⨟ σ        ≡ σ
(sub-idR)   σ ⨟ ids        ≡ σ
(sub-assoc) (σ ⨟ τ) ⨟ θ    ≡ σ ⨟ (τ ⨟ θ)
(sub-dist)  (M • σ) ⨟ τ    ≡ (⟪ τ ⟫ M) • (σ ⨟ τ)
```

The first group of equations describe how the `•` operator acts like cons. The equation `sub-head` says that the variable zero `Z` returns the head of the sequence (it acts like the `car` of Lisp). Similarly, `sub-tail` says that sequencing with shift `↑` returns the tail of the sequence (it acts like `cdr` of Lisp). The `sub-η` equation is the η-expansion rule for sequences, saying that taking the head and tail of a sequence, and then cons'ing them together yields the original sequence. The `Z-shift` equation says that cons'ing zero onto the shifted sequence produces the identity sequence.

The next four equations involve applying substitutions to terms. The equation `sub-id` says that the identity substitution returns the term unchanged. The equations `sub-app` and `sub-abs` says that substitution is a congruence for the lambda calculus. The `sub-sub` equation says that the sequence operator `⨟` behaves as intended.

The last four equations concern the sequencing of substitutions. The first two equations, `sub-idL` and `sub-idR`, say that `ids` is the left and right unit of the sequencing operator. The `sub-assoc` equation says that sequencing is associative. Finally, `sub-dist` says that post-sequencing distributes through cons.

Relating the σ algebra and substitution functions

The definitions of substitution `N [ M ]` and parallel substitution `subst σ N` depend on several auxiliary functions: `rename`, `exts`, `ext`, and `subst-zero`. We shall relate those functions to terms in the σ algebra.

To begin with, renaming can be expressed in terms of substitution. We have

```
rename ρ M ≡ ⟪ ren ρ ⟫ M                (rename-subst-ren)
```

where `ren` turns a renaming `ρ` into a substitution by post-composing `ρ` with the identity substitution.

```
ren : ∀{Γ Δ} → Rename Γ Δ → Subst Γ Δ
ren ρ = ids ∘ ρ
```

When the renaming is the increment function, then it is equivalent to shift.

```
ren S_ ≡ ↑                              (ren-shift)

rename S_ M ≡ ⟪ ↑ ⟫ M                   (rename-shift)
```

Renaming with the identity renaming leaves the term unchanged.

```
rename (λ {A} x → x) M ≡ M              (rename-id)
```

Next we relate the `exts` function to the σ algebra. Recall that the `exts` function extends a substitution as follows:

```
exts σ = ` Z, rename S_ (σ 0), rename S_ (σ 1), rename S_ (σ 2), ...
```

So `exts` is equivalent to cons'ing Z onto the sequence formed by applying `σ` and then shifting.

```
exts σ ≡ ` Z • (σ ⨟ ↑)                 (exts-cons-shift)
```

The `ext` function does the same job as `exts` but for renamings instead of substitutions. So composing `ext` with `ren` is the same as composing `ren` with `exts`.

```
ren (ext ρ) ≡ exts (ren ρ)            (ren-ext)
```

Thus, we can recast the `exts-cons-shift` equation in terms of renamings.

```
ren (ext ρ) ≡ ` Z • (ren ρ ⨟ ↑)       (ext-cons-Z-shift)
```

It is also useful to specialize the `sub-sub` equation of the σ algebra to the situation where the first substitution is a renaming.

```
⟪ σ ⟫ (rename ρ M) ≡ ⟪ σ ∘ ρ ⟫ M       (rename-subst)
```

The `subst-zero M` substitution is equivalent to cons'ing `M` onto the identity substitution.

```
subst-zero M ≡ M • ids                (subst-Z-cons-ids)
```

Finally, sequencing `exts σ` with `subst-zero M` is equivalent to cons'ing `M` onto `σ`.

```
exts σ ⨟ subst-zero M ≡ (M • σ)       (subst-zero-exts-cons)
```

Proofs of sub-head, sub-tail, sub-η, Z-shift, sub-idL, sub-dist, and sub-app

We start with the proofs that are immediate from the definitions of the operators.

```
sub-head : ∀ {Γ Δ} {A} {M : Δ ⊢ A}{σ : Subst Γ Δ}
         → ⟪ M • σ ⟫ (` Z) ≡ M
sub-head = refl
```

```
sub-tail : ∀{Γ Δ} {A B} {M : Δ ⊢ A} {σ : Subst Γ Δ}
         → (↑ ⨟ M • σ) {A = B} ≡ σ
sub-tail = extensionality λ x → refl
```

```
sub-η : ∀{Γ Δ} {A B} {σ : Subst (Γ , A) Δ}
      → (⟪ σ ⟫ (` Z) • (↑ ⨟ σ)) {A = B} ≡ σ
sub-η {Γ}{Δ}{A}{B}{σ} = extensionality λ x → lemma
  where
  lemma : ∀ {x} → ((⟪ σ ⟫ (` Z)) • (↑ ⨟ σ)) x ≡ σ x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

```
Z-shift : ∀{Γ}{A B}
        → ((` Z) • ↑) ≡ ids {Γ , A} {B}
Z-shift {Γ}{A}{B} = extensionality lemma
  where
  lemma : (x : Γ , A ∋ B) → ((` Z) • ↑) x ≡ ids x
  lemma Z = refl
  lemma (S y) = refl
```

```
sub-idL : ∀{Γ Δ} {σ : Subst Γ Δ} {A}
  → ids ⨟ σ ≡ σ {A}
sub-idL = extensionality λ x → refl
```

```
sub-dist : ∀{Γ Δ Σ : Context} {A B} {σ : Subst Γ Δ} {τ : Subst Δ Σ}
             {M : Δ ⊢ A}
         → ((M • σ) ⨟ τ) ≡ ((subst τ M) • (σ ⨟ τ)) {B}
sub-dist {Γ}{Δ}{Σ}{A}{B}{σ}{τ}{M} = extensionality λ x → lemma {x = x}
  where
  lemma : ∀ {x : Γ , A ∋ B} → ((M • σ) ⨟ τ) x ≡ ((subst τ M) • (σ ⨟ τ)) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

```
sub-app : ∀{Γ Δ} {σ : Subst Γ Δ} {L : Γ ⊢ ★}{M : Γ ⊢ ★}
        → ⟪ σ ⟫ (L · M) ≡ (⟪ σ ⟫ L) · (⟪ σ ⟫ M)
sub-app = refl
```

Interlude: congruences

In this section we establish congruence rules for the σ algebra operators `•` and `⨟` and for `subst` and its helper functions `ext`, `rename`, `exts`, and `subst-zero`. These congruence rules help with the equational reasoning in the later sections of this chapter.

[JGS: I would have liked to prove all of these via cong and cong₂, but I have not yet found a way to make that work. It seems that various implicit parameters get in the way.]

```
cong-ext : ∀{Γ Δ}{ρ ρ′ : Rename Γ Δ}{B}
  → (∀{A} → ρ ≡ ρ′ {A})
    ---------------------------------
  → ∀{A} → ext ρ {B = B} ≡ ext ρ′ {A}
cong-ext{Γ}{Δ}{ρ}{ρ′}{B} rr {A} = extensionality λ x → lemma {x}
  where
  lemma : ∀{x : Γ , B ∋ A} → ext ρ x ≡ ext ρ′ x
  lemma {Z} = refl
  lemma {S y} = cong S_ (cong-app rr y)
```

```
cong-rename : ∀{Γ Δ}{ρ ρ′ : Rename Γ Δ}{B}{M : Γ ⊢ B}
  → (∀{A} → ρ ≡ ρ′ {A})
    ------------------------
  → rename ρ M ≡ rename ρ′ M
cong-rename {M = ` x} rr = cong `_ (cong-app rr x)
cong-rename {ρ = ρ} {ρ′ = ρ′} {M = ƛ N} rr =
  cong ƛ_ (cong-rename {ρ = ext ρ}{ρ′ = ext ρ′}{M = N} (cong-ext rr))
cong-rename {M = L · M} rr =
  cong₂ _·_ (cong-rename rr) (cong-rename rr)
```

```
cong-exts : ∀{Γ Δ}{σ σ′ : Subst Γ Δ}{B}
  → (∀{A} → σ ≡ σ′ {A})
    -----------------------------------
  → ∀{A} → exts σ {B = B} ≡ exts σ′ {A}
cong-exts{Γ}{Δ}{σ}{σ′}{B} ss {A} = extensionality λ x → lemma {x}
```

```
  where
  lemma : ∀{x} → exts σ x ≡ exts σ′ x
  lemma {Z} = refl
  lemma {S x} = cong (rename S_) (cong-app (ss {A}) x)
```

```
cong-sub : ∀{Γ Δ}{σ σ′ : Subst Γ Δ}{A}{M M′ : Γ ⊢ A}
            → (∀{A} → σ ≡ σ′ {A}) → M ≡ M′
              ------------------------------
            → subst σ M ≡ subst σ′ M′
cong-sub {Γ} {Δ} {σ} {σ′} {A} {` x} ss refl = cong-app ss x
cong-sub {Γ} {Δ} {σ} {σ′} {A} {ƛ M} ss refl =
  cong ƛ_ (cong-sub {σ = exts σ}{σ′ = exts σ′} {M = M} (cong-exts ss) refl)
cong-sub {Γ} {Δ} {σ} {σ′} {A} {L · M} ss refl =
  cong₂ _·_ (cong-sub {M = L} ss refl) (cong-sub {M = M} ss refl)
```

```
cong-sub-zero : ∀{Γ}{B : Type}{M M′ : Γ ⊢ B}
  → M ≡ M′
    -----------------------------------------
  → ∀{A} → subst-zero M ≡ (subst-zero M′) {A}
cong-sub-zero {Γ}{B}{M}{M′} mm' {A} =
  extensionality λ x → cong (λ z → subst-zero z x) mm'
```

```
cong-cons : ∀{Γ Δ}{A}{M N : Δ ⊢ A}{σ τ : Subst Γ Δ}
  → M ≡ N → (∀{A} → σ {A} ≡ τ {A})
    --------------------------------
  → ∀{A} → (M • σ) {A} ≡ (N • τ) {A}
cong-cons{Γ}{Δ}{A}{M}{N}{σ}{τ} refl st {A′} = extensionality lemma
  where
  lemma : (x : Γ , A ∋ A′) → (M • σ) x ≡ (M • τ) x
  lemma Z = refl
  lemma (S x) = cong-app st x
```

```
cong-seq : ∀{Γ Δ Σ}{σ σ′ : Subst Γ Δ}{τ τ′ : Subst Δ Σ}
  → (∀{A} → σ {A} ≡ σ′ {A}) → (∀{A} → τ {A} ≡ τ′ {A})
  → ∀{A} → (σ ⨟ τ) {A} ≡ (σ′ ⨟ τ′) {A}
cong-seq {Γ}{Δ}{Σ}{σ}{σ′}{τ}{τ′} ss' tt' {A} = extensionality lemma
  where
  lemma : (x : Γ ∋ A) → (σ ⨟ τ) x ≡ (σ′ ⨟ τ′) x
  lemma x =
    begin
      (σ ⨟ τ) x
    ≡⟨⟩
      subst τ (σ x)
    ≡⟨ cong (subst τ) (cong-app ss' x) ⟩
      subst τ (σ′ x)
    ≡⟨ cong-sub{M = σ′ x} tt' refl ⟩
      subst τ′ (σ′ x)
    ≡⟨⟩
      (σ′ ⨟ τ′) x
    ∎
```

Relating `rename`, `exts`, `ext`, and `subst-zero` to the σ algebra

In this section we establish equations that relate `subst` and its helper functions (`rename`, `exts`, `ext`, and `subst-zero`) to terms in the σ algebra.

The first equation we prove is

```
rename ρ M ≡ ⟪ ren ρ ⟫ M                (rename-subst-ren)
```

Because `subst` uses the `exts` function, we need the following lemma which says that `exts` and `ext` do the same thing except that `ext` works on renamings and `exts` works on substitutions.

```
ren-ext : ∀ {Γ Δ}{B C : Type} {ρ : Rename Γ Δ}
        → ren (ext ρ {B = B}) ≡ exts (ren ρ) {C}
ren-ext {Γ}{Δ}{B}{C}{ρ} = extensionality λ x → lemma {x = x}
  where
  lemma : ∀ {x : Γ , B ∋ C} → (ren (ext ρ)) x ≡ exts (ren ρ) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

With this lemma in hand, the proof is a straightforward induction on the term `M`.

```
rename-subst-ren : ∀ {Γ Δ}{A} {ρ : Rename Γ Δ}{M : Γ ⊢ A}
                 → rename ρ M ≡ ⟪ ren ρ ⟫ M
rename-subst-ren {M = ` x} = refl
rename-subst-ren {ρ = ρ}{M = ƛ N} =
  begin
      rename ρ (ƛ N)
    ≡⟨⟩
      ƛ rename (ext ρ) N
    ≡⟨ cong ƛ_ (rename-subst-ren {ρ = ext ρ}{M = N}) ⟩
      ƛ ⟪ ren (ext ρ) ⟫ N
    ≡⟨ cong ƛ_ (cong-sub {M = N} ren-ext refl) ⟩
      ƛ ⟪ exts (ren ρ) ⟫ N
    ≡⟨⟩
      ⟪ ren ρ ⟫ (ƛ N)
  ∎
rename-subst-ren {M = L · M} = cong₂ _·_ rename-subst-ren rename-subst-ren
```

The substitution `ren S_` is equivalent to `↑`.

```
ren-shift : ∀{Γ}{A}{B}
          → ren S_ ≡ ↑ {A = B} {A}
ren-shift {Γ}{A}{B} = extensionality λ x → lemma {x = x}
  where
  lemma : ∀ {x : Γ ∋ A} → ren (S_{B = B}) x ≡ ↑ {A = B} x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

The substitution `rename S_ M` is equivalent to shifting: `⟪ ↑ ⟫ M`.

```
rename-shift : ∀{Γ} {A} {B} {M : Γ ⊢ A}
             → rename (S_{B = B}) M ≡ ⟪ ↑ ⟫ M
rename-shift{Γ}{A}{B}{M} =
  begin
```

```
    rename S_ M
  ≡⟨ rename-subst-ren ⟩
    ⟪ ren S_ ⟫ M
  ≡⟨ cong-sub{M = M} ren-shift refl ⟩
    ⟪ ↑ ⟫ M
  ∎
```

Next we prove the equation `exts-cons-shift`, which states that `exts` is equivalent to cons'ing Z onto the sequence formed by applying `σ` and then shifting. The proof is by case analysis on the variable `x`, using `rename-subst-ren` for when `x = S y`.

```
exts-cons-shift : ∀{Γ Δ} {A B} {σ : Subst Γ Δ}
                → exts σ {A} {B} ≡ (` Z • (σ ⨟ ↑))
exts-cons-shift = extensionality λ x → lemma{x = x}
  where
  lemma : ∀{Γ Δ} {A B} {σ : Subst Γ Δ} {x : Γ , B ∋ A}
                 → exts σ x ≡ (` Z • (σ ⨟ ↑)) x
  lemma {x = Z} = refl
  lemma {x = S y} = rename-subst-ren
```

As a corollary, we have a similar correspondence for `ren (ext ρ)`.

```
ext-cons-Z-shift : ∀{Γ Δ} {ρ : Rename Γ Δ}{A}{B}
                 → ren (ext ρ {B = B}) ≡ (` Z • (ren ρ ⨟ ↑)) {A}
ext-cons-Z-shift {Γ}{Δ}{ρ}{A}{B} =
  begin
    ren (ext ρ)
  ≡⟨ ren-ext ⟩
    exts (ren ρ)
  ≡⟨ exts-cons-shift{σ = ren ρ} ⟩
    ((` Z) • (ren ρ ⨟ ↑))
  ∎
```

Finally, the `subst-zero M` substitution is equivalent to cons'ing `M` onto the identity substitution.

```
subst-Z-cons-ids : ∀{Γ}{A B : Type}{M : Γ ⊢ B}
                 → subst-zero M ≡ (M • ids) {A}
subst-Z-cons-ids = extensionality λ x → lemma {x = x}
  where
  lemma : ∀{Γ}{A B : Type}{M : Γ ⊢ B}{x : Γ , B ∋ A}
                      → subst-zero M x ≡ (M • ids) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

Proofs of sub-abs, sub-id, and rename-id

The equation `sub-abs` follows immediately from the equation `exts-cons-shift`.

```
sub-abs : ∀{Γ Δ} {σ : Subst Γ Δ} {N : Γ , ★ ⊢ ★}
        → ⟪ σ ⟫ (ƛ N) ≡ ƛ ⟪ (` Z) • (σ ⨟ ↑) ⟫ N
sub-abs {σ = σ}{N = N} =
  begin
    ⟪ σ ⟫ (ƛ N)
  ≡⟨⟩
    ƛ ⟪ exts σ ⟫ N
  ≡⟨ cong ƛ_ (cong-sub{M = N} exts-cons-shift refl) ⟩
    ƛ ⟪ (` Z) • (σ ⨟ ↑) ⟫ N
  ∎
```

The proof of `sub-id` requires the following lemma which says that extending the identity substitution produces the identity substitution.

```
exts-ids : ∀{Γ}{A B}
         → exts ids ≡ ids {Γ , B} {A}
exts-ids {Γ}{A}{B} = extensionality lemma
  where lemma : (x : Γ , B ∋ A) → exts ids x ≡ ids x
        lemma Z = refl
        lemma (S x) = refl
```

The proof of `⟪ ids ⟫ M ≡ M` now follows easily by induction on `M`, using `exts-ids` in the case for `M ≡ ƛ N`.

```
sub-id : ∀{Γ} {A} {M : Γ ⊢ A}
         → ⟪ ids ⟫ M ≡ M
sub-id {M = ` x} = refl
sub-id {M = ƛ N} =
  begin
    ⟪ ids ⟫ (ƛ N)
  ≡⟨⟩
    ƛ ⟪ exts ids ⟫ N
  ≡⟨ cong ƛ_ (cong-sub{M = N} exts-ids refl) ⟩
    ƛ ⟪ ids ⟫ N
  ≡⟨ cong ƛ_ sub-id ⟩
    ƛ N
  ∎
sub-id {M = L · M} = cong₂ _·_ sub-id sub-id
```

The `rename-id` equation is a corollary is `sub-id`.

```
rename-id : ∀ {Γ}{A} {M : Γ ⊢ A}
  → rename (λ {A} x → x) M ≡ M
rename-id {M = M} =
  begin
    rename (λ {A} x → x) M
  ≡⟨ rename-subst-ren ⟩
    ⟪ ren (λ {A} x → x) ⟫ M
  ≡⟨⟩
    ⟪ ids ⟫ M
  ≡⟨ sub-id ⟩
    M
  ∎
```

Proof of sub-idR

The proof of `sub-idR` follows directly from `sub-id`.

```
sub-idR : ∀{Γ Δ} {σ : Subst Γ Δ} {A}
  → (σ ⨟ ids) ≡ σ {A}
sub-idR {Γ}{σ = σ}{A} =
          begin
            σ ⨟ ids
          ≡⟨⟩
            ⟪ ids ⟫ ∘ σ
          ≡⟨ extensionality (λ x → sub-id) ⟩
            σ
          ∎
```

Proof of sub-sub

The `sub-sub` equation states that sequenced substitutions `σ ⨟ τ` are equivalent to first applying `σ` then applying `τ`.

```
⟪ τ ⟫ ⟪ σ ⟫ M  ≡ ⟪ σ ⨟ τ ⟫ M
```

The proof requires several lemmas. First, we need to prove the specialization for renaming.

```
rename ρ (rename ρ′ M) ≡ rename (ρ ∘ ρ′) M
```

This in turn requires the following lemma about `ext`.

```
compose-ext : ∀{Γ Δ Σ}{ρ : Rename Δ Σ} {ρ′ : Rename Γ Δ} {A B}
            → ((ext ρ) ∘ (ext ρ′)) ≡ ext (ρ ∘ ρ′) {B} {A}
compose-ext = extensionality λ x → lemma {x = x}
  where
  lemma : ∀{Γ Δ Σ}{ρ : Rename Δ Σ} {ρ′ : Rename Γ Δ} {A B} {x : Γ , B ∋ A}
              → ((ext ρ) ∘ (ext ρ′)) x ≡ ext (ρ ∘ ρ′) x
  lemma {x = Z} = refl
  lemma {x = S x} = refl
```

To prove that composing renamings is equivalent to applying one after the other using `rename`, we proceed by induction on the term `M`, using the `compose-ext` lemma in the case for `M ≡ ƛ N`.

```
compose-rename : ∀{Γ Δ Σ}{A}{M : Γ ⊢ A}{ρ : Rename Δ Σ}{ρ′ : Rename Γ Δ}
  → rename ρ (rename ρ′ M) ≡ rename (ρ ∘ ρ′) M
compose-rename {M = ` x} = refl
compose-rename {Γ}{Δ}{Σ}{A}{ƛ N}{ρ}{ρ′} = cong ƛ_ G
  where
  G : rename (ext ρ) (rename (ext ρ′) N) ≡ rename (ext (ρ ∘ ρ′)) N
  G =
      begin
        rename (ext ρ) (rename (ext ρ′) N)
      ≡⟨ compose-rename{ρ = ext ρ}{ρ′ = ext ρ′} ⟩
        rename ((ext ρ) ∘ (ext ρ′)) N
      ≡⟨ cong-rename compose-ext ⟩
        rename (ext (ρ ∘ ρ′)) N
      ∎
compose-rename {M = L · M} = cong₂ _·_ compose-rename compose-rename
```

The next lemma states that if a renaming and substitution commute on variables, then they also commute on terms. We explain the proof in detail below.

```
commute-subst-rename : ∀{Γ Δ}{M : Γ ⊢ ★}{σ : Subst Γ Δ}
                             {ρ : ∀{Γ} → Rename Γ (Γ , ★)}
    → (∀{x : Γ ∋ ★} → exts σ {B = ★} (ρ x) ≡ rename ρ (σ x))
    → subst (exts σ {B = ★}) (rename ρ M) ≡ rename ρ (subst σ M)
commute-subst-rename {M = ` x} r = r
commute-subst-rename{Γ}{Δ}{ƛ N}{σ}{ρ} r =
  cong ƛ_ (commute-subst-rename{Γ , ★}{Δ , ★}{N}
           {exts σ}{ρ = ρ′} (λ {x} → H {x}))
  where
  ρ′ : ∀ {Γ} → Rename Γ (Γ , ★)
  ρ′ {∅} = λ ()
  ρ′ {Γ , ★} = ext ρ

  H : {x : Γ , ★ ∋ ★} → exts (exts σ) (ext ρ x) ≡ rename (ext ρ) (exts σ x)
  H {Z} = refl
  H {S y} =
    begin
      exts (exts σ) (ext ρ (S y))
    ≡⟨⟩
      rename S_ (exts σ (ρ y))
    ≡⟨ cong (rename S_) r ⟩
      rename S_ (rename ρ (σ y))
    ≡⟨ compose-rename ⟩
      rename (S_ ∘ ρ) (σ y)
    ≡⟨ cong-rename refl ⟩
      rename ((ext ρ) ∘ S_) (σ y)
    ≡⟨ sym compose-rename ⟩
      rename (ext ρ) (rename S_ (σ y))
    ≡⟨⟩
      rename (ext ρ) (exts σ (S y))
    ∎
commute-subst-rename {M = L · M}{ρ = ρ} r =
  cong₂ _·_ (commute-subst-rename{M = L}{ρ = ρ} r)
            (commute-subst-rename{M = M}{ρ = ρ} r)
```

The proof is by induction on the term `M`.

- If `M` is a variable, then we use the premise to conclude.
- If `M ≡ ƛ N`, we conclude using the induction hypothesis for `N`. However, to use the induction hypothesis, we must show that

  ```
      exts (exts σ) (ext ρ x) ≡ rename (ext ρ) (exts σ x)
  ```

  We prove this equation by cases on x.

  - If `x = Z`, the two sides are equal by definition.
  - If `x = S y`, we obtain the goal by the following equational reasoning.

    ```
        exts (exts σ) (ext ρ (S y))
      ≡ rename S_ (exts σ (ρ y))
      ≡ rename S_ (rename S_ (σ (ρ y)      (by the premise)
      ≡ rename (ext ρ) (exts σ (S y))      (by compose-rename)
      ≡ rename ((ext ρ) ∘ S_) (σ y)
      ≡ rename (ext ρ) (rename S_ (σ y))   (by compose-rename)
      ≡ rename (ext ρ) (exts σ (S y))
    ```

- If `M` is an application, we obtain the goal using the induction hypothesis for each subterm.

The last lemma needed to prove `sub-sub` states that the `exts` function distributes with sequencing. It is a corollary of `commute-subst-rename` as described below. (It would have been nicer to prove this directly by equational reasoning in the σ algebra, but that would require the `sub-assoc` equation, whose proof depends on `sub-sub`, which in turn depends on this lemma.)

```
exts-seq : ∀{Γ Δ Δ′} {σ₁ : Subst Γ Δ} {σ₂ : Subst Δ Δ′}
         → ∀ {A} → (exts σ₁ ⨟ exts σ₂) {A} ≡ exts (σ₁ ⨟ σ₂)
exts-seq = extensionality λ x → lemma {x = x}
  where
  lemma : ∀{Γ Δ Δ′}{A}{x : Γ , ★ ∋ A} {σ₁ : Subst Γ Δ}{σ₂ : Subst Δ Δ′}
    → (exts σ₁ ⨟ exts σ₂) x ≡ exts (σ₁ ⨟ σ₂) x
  lemma {x = Z} = refl
  lemma {A = ★}{x = S x}{σ₁}{σ₂} =
    begin
      (exts σ₁ ⨟ exts σ₂) (S x)
    ≡⟨⟩
      ⟪ exts σ₂ ⟫ (rename S_ (σ₁ x))
    ≡⟨ commute-subst-rename{M = σ₁ x}{σ = σ₂}{ρ = S_} refl ⟩
      rename S_ (⟪ σ₂ ⟫ (σ₁ x))
    ≡⟨⟩
      rename S_ ((σ₁ ⨟ σ₂) x)
    ∎
```

The proof proceed by cases on `x`.

- If `x = Z`, the two sides are equal by the definition of `exts` and sequencing.
- If `x = S x`, we unfold the first use of `exts` and sequencing, then apply the lemma `commute-subst-rename`. We conclude by the definition of sequencing.

Now we come to the proof of `sub-sub`, which we explain below.

```
sub-sub : ∀{Γ Δ Σ}{A}{M : Γ ⊢ A} {σ₁ : Subst Γ Δ}{σ₂ : Subst Δ Σ}
            → ⟪ σ₂ ⟫ (⟪ σ₁ ⟫ M) ≡ ⟪ σ₁ ⨟ σ₂ ⟫ M
sub-sub {M = ` x} = refl
sub-sub {Γ}{Δ}{Σ}{A}{ƛ N}{σ₁}{σ₂} =
  begin
    ⟪ σ₂ ⟫ (⟪ σ₁ ⟫ (ƛ N))
  ≡⟨⟩
    ƛ ⟪ exts σ₂ ⟫ (⟪ exts σ₁ ⟫ N)
  ≡⟨ cong ƛ_ (sub-sub{M = N}{σ₁ = exts σ₁}{σ₂ = exts σ₂}) ⟩
    ƛ ⟪ exts σ₁ ⨟ exts σ₂ ⟫ N
  ≡⟨ cong ƛ_ (cong-sub{M = N} (λ {A} → exts-seq) refl) ⟩
    ƛ ⟪ exts ( σ₁ ⨟ σ₂) ⟫ N
  ∎
sub-sub {M = L · M} = cong₂ _·_ (sub-sub{M = L}) (sub-sub{M = M})
```

We proceed by induction on the term `M`.

- If `M = x`, then both sides are equal to `σ₂ (σ₁ x)`.
- If `M = ƛ N`, we first use the induction hypothesis to show that

  ```
      ƛ ⟪ exts σ₂ ⟫ (⟪ exts σ₁ ⟫ N) ≡ ƛ ⟪ exts σ₁ ⨟ exts σ₂ ⟫ N
  ```

  and then use the lemma `exts-seq` to show

```
    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

[illegible]

Appendix B

Acknowledgements

Thank you to:

- The inventors of Agda, for a new playground.
- The authors of Software Foundations, for inspiration.

A special thank you, for inventing ideas on which this book is based, and for hand-holding:

Andreas Abel

Catarina Coquand

Thierry Coquand

David Darais

Per Martin-Löf

Lena Magnusson

Conor McBride

James McKinna

Ulf Norell

For pull requests big and small, and for answering questions on the Agda mailing list:

Marko Dimjašević

Zbigniew Stanasiuk

Reza Gharibi

Yasu Watanabe

Michael Reed

Chad Nester

Fangyi Zhou

Mo Mirza

Juhana Laurinharju

Qais Patankar

Orestis Melkonian

Kenichi Asai

phi16

Pedro Minicz

Jonathan Prieto

Alexandru Brisan

Sebastian Miele

bc^2

Murilo Giacometti Rocha

Michel Steuwer

Matthew Healy

Lorenzo Martinico

caryoscelus

Zach Brown

Syed Turab Ali Jafri

Spencer Whitt

Peter Thiemann

Alexandre Moreno

Ingo Blechschmidt

G. Allais

Matthias Gabriel

Isaac Elliott

Liang-Ting Chen

Mike He

Zack Grannan

Nicolas Wu

Slava

Vikraman Choudhury

Stephan Boyer

Amr Sabry

Rodrigo Bernardo

purchan

Nathaniel Carroll

Phil de Joux

N. Raghavendra

Léo Gillot-Lamure

Nils Anders Danielsson

Miëtek Bak

Merlin Göttlinger

Liam O’Connor

James Wood

Kenneth MacKenzie

Stefan Kranich

koo5

Kartik Singhal

John Maraist

Hugo Gualandi

Gan Shen

Georgi Lyubenov

Gergő Érdi

Roman Kireev

David Janin

Deniz Alp

April Gonçalves

citrusmunch

Chike Abuah

Ben Darwin

Anish Tondwalkar

Adam Sandberg Eriksson

Ulf Norell

Torsten Grust

Oling Cat

Gagan Devagiri

Dee Yeum

András Kovács

[Your name goes here]

For contributions to the answers repository:

William Cook

David Banas

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

For support:

- EPSRC Programme Grant EP/K034413/1
- NSF Grant No. 1814460
- Foundation Sciences Mathematiques de Paris (FSMP) Distinguised Professor Fellowship

Appendix C

Fonts

```
module plfa.backmatter.Fonts where
```

Preferably, all vertical bars should line up.

```
--------------------------|
abcdefghijklmnopqrstuvwxyz|
ABCDEFGHIJKLMNOPQRSTUVWXYZ|
ᵃᵇᶜᵈᵉᶠᵍʰⁱʲᵏˡᵐⁿᵒᵖ ʳˢᵗᵘᵛʷˣʸᶻ|
ᴬᴮ ᴰᴱ ᴳᴴᴵᴶᴷᴸᴹᴺᴼᴾ ᴿ ᵀᵁⱽᵂ   |
ₐ   ₑ  ₕᵢⱼₖₗₘₙₒₚ ᵣₛₜᵤ  ₓ   |
--------------------------|
----------|
0123456789|
⁰¹²³⁴⁵⁶⁷⁸⁹|
₀₁₂₃₄₅₆₇₈₉|
----------|
------------------------|
αβγδεζηθικλμνξοπρστυφχψω|
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ|
------------------------|
----|
≠≠≠≠|
ηημμ|
ΓΓΔΔ|
ΣΣΠΠ|
λλλλ|
ƛƛƛƛ|
····|
××××|
ℓℓℓℓ|
≡≡≡≡|
¬¬¬¬|
≤≤≥≥|
∅∅∅∅|
————|
††‡‡|
^^^^|
''""|
``~~|
⊎⊎⊃⊃|
∧∧∨∨|
⊗⊗⊗⊗|
⊔⊔⊔⊔|
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|
----------|
```