This is the Project Title

Your Name

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2019

Abstract

This is an example of `infthesis` style. The file `skeleton.tex` generates this document and can be used to get a "skeleton" for your thesis. The abstract should summarise your report and fit in the space on the first page. You may, of course, use any other software to write your report, as long as you follow the same style. That means: producing a title page as given here, and including a table of contents and bibliography.

Acknowledgements

Acknowledgements go here.

Contents

Chapter 1

Introduction

The document structure should include:

- The title page in the format used above.
- An optional acknowledgements page.
- The table of contents.
- The report text divided into chapters as appropriate.
- The bibliography.

Commands for generating the title page appear in the skeleton file and are self explanatory. The file also includes commands to choose your report type (project report, thesis or dissertation) and degree. These will be placed in the appropriate place in the title page.

The default behaviour of the documentclass is to produce documents typeset in 12 point. Regardless of the formatting system you use, it is recommended that you submit your thesis printed (or copied) double sided.

The report should be printed single-spaced. It should be 30 to 60 pages long, and preferably no shorter than 20 pages. Appendices are in addition to this and you should place detail here which may be too much or not strictly necessary when reading the relevant section.

1.1 Using Sections

Divide your chapters into sub-parts as appropriate.

1.2 Citations

Note that citations (like [?] or [?]) can be generated using `BibTeX` or by using the `thebibliography` environment. This makes sure that the table of contents includes an entry for the bibliography. Of course you may use any other method as well.

1.3 Options

There are various documentclass options, see the documentation. Here we are using an option (`bsc` or `minf`) to choose the degree type, plus:

- `frontabs` (recommended) to put the abstract on the front page;
- `twoside` (recommended) to format for two-sided printing, with each chapter starting on a right-hand page;
- `singlespacing` (required) for single-spaced formating; and
- `parskip` (a matter of taste) which alters the paragraph formatting so that paragraphs are separated by a vertical space, and there is no indentation at the start of each paragraph.

Chapter 2

The Real Thing

2.1 Imports

```
module PCF where

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc)
open import Relation.Nullary using (¬_)
```

2.2 Syntax

```
infix 4 _⊢_
infix 4 _∋_
infixl 5 _,_

infix 5 ƛ_
infixl 7 _·_
infix 8 `suc_
infix 9 `_
infix 9 S_
infix 9 #_

infixr 7 _⇒_
```

2.3 Types

```
data Type : Set where
  `ℕ    : Type
  _⇒_ : Type → Type → Type
```

2.4 Contexts

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

2.5 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
```

2.6 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 , 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
```

2.7 Abbreviating de Bruijn indices

```
lookup : Context → ℕ → Type
lookup (Γ , A) zero    = A
lookup (Γ , _) (suc n) = lookup Γ n
lookup ∅ _             = ⊥-elim impossible
  where postulate impossible : ⊥

count : ∀ {Γ} → (n : ℕ) → Γ ∋ lookup Γ n
count {Γ , _} zero    = Z
count {Γ , _} (suc n) = S (count n)
count {∅} _           = ⊥-elim impossible
  where postulate impossible : ⊥

#_ : ∀ {Γ} → (n : ℕ) → Γ ⊢ lookup Γ n
# n = ' count n
```

2.8 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)

extλ : ∀ {Γ Δ} → (∀ {A} → Γ ∋ A → Δ ∋ A) → (∀ {A B C} → Γ , A , B ∋ C → Δ , A , B ∋ C)
extλ ρ Z         = Z
extλ ρ (S Z)     = S Z
extλ ρ (S S x) = S (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 (extƛ 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)
```

2.9 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)

extsƛ : ∀ {Γ Δ} → (∀ {A} → Γ ∋ A → Δ ⊢ A) → (∀ {A B C} → Γ , A , B ∋ C → Δ , A , B ⊢ C)
extsƛ σ Z        = ` Z
extsƛ σ (S Z)    = ` S Z
extsƛ σ (S S x) = rename (λ v → S S v) (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 (extsƛ 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)
```

2.10 Single and double substitution

```
_[_] : ∀ {Γ A B}
  → Γ , A ⊢ B
  → Γ ⊢ A
  ------------
  → Γ ⊢ B
_[_] {Γ} {A} N V = subst {Γ , A} {Γ} σ N
  where
  σ : ∀ {B} → Γ , A ∋ B → Γ ⊢ B
  σ Z      = V
  σ (S x) = ` x

_[_][_] : ∀ {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
```

2.11 Values

```
  data Value : ∀ {Γ A} → Γ ⊢ A → Set where

    -- functions

    V-λ : ∀ {Γ A B} {N : Γ , A ⇒ B , A ⊢ B}
        ---------------------------
      → Value (λ N)

    -- naturals

    V-zero : ∀ {Γ} →
        -----------------
        Value (`zero {Γ})

    V-suc_ : ∀ {Γ} {V : Γ ⊢ `ℕ}
      → Value V
        --------------
      → Value (`suc V)
```

2.12 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-λ : ∀ {Γ A B} {N : Γ , A ⇒ B , A ⊢ B} {V : Γ ⊢ A} -- TODO
    → Value V
      ---------------------
    → (λ N) · V —→ N [ λ 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 ]
```

2.13 Reflexive and transitive closure

```
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
```

2.14 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)
```

2.15 Evaluation

```
data Gas : Set where
  gas : ℕ → Gas

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 : ∀ {A} {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
```

2.16 Examples

```
two : ∀ {Γ} → Γ ⊢ `ℕ
two = `suc `suc `zero

plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = ƛ ƛ (case (# 2) (# 0) (`suc (# 4 · # 0 · # 1)))

2+2 : ∅ ⊢ `ℕ
2+2 = plus · two · two
```

2.17 Imports

```
module Closure where

import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc)
open import Relation.Nullary using (¬_)
open import Data.List using (List ; _::_ ; [])
```

2.18 Syntax

```
infix 4 _⊢_
infix 4 _∋_
infix 5 ⟨⟨_,_⟩⟩
infixr 9 s_

infixr 7 _⇒_

infixl 7 _·_
infix 8 ‘suc_
infix 9 ‘_
infix 9 #_
```

2.19 Types

```
data Type : Set where
  ‘ℕ   : Type
  _⇒_ : Type → Type → Type
```

2.20 Contexts

```
Context : Set
Context = List Type
```

2.21 Variables and the lookup judgment

```
data _∋_ : Context → Type → Set where

  z : ∀ {Γ A}
      ----------
    → A :: Γ ∋ A

  s_ : ∀ {Γ A B}
    → Γ ∋ B
      ----------
    → A :: Γ ∋ B
```

2.22 Terms, enviroments, and the typing judgment

```
data _⊢_ : Context → Type → Set

data Env : Context → Context → Set where
  []   : ∀ {Γ} → Env [] Γ
  _::_ : ∀ {Γ Δ A} → Γ ⊢ A → Env Δ Γ → Env (A :: Δ) Γ

data _⊢_ where

  -- variables

  `_      : ∀ {Γ A}
          → Γ ∋ A
            ----------
          → Γ ⊢ A

  -- functions

  _·_     : ∀ {Γ A B}
          → Γ ⊢ (A ⇒ B)
          → Γ ⊢ A
            ----------------
          → Γ ⊢ B

  -- closures

  ⟨⟨_,_⟩⟩ : ∀ {Γ Δ A B}
          → A :: A ⇒ B :: Δ ⊢ B
          → Env Δ Γ -- Γ ⊢ Context→Env Δ
```

```
      ------------
    → Γ ⊢ (A ⇒ B)

-- naturals

`zero : ∀ {Γ}
    ---------
  → Γ ⊢ `ℕ

`suc_ : ∀ {Γ}
  → Γ ⊢ `ℕ
    ------------
  → Γ ⊢ `ℕ

case : ∀ {Γ A}
  → Γ ⊢ `ℕ
  → Γ ⊢ A
  → `ℕ :: Γ ⊢ A
    ----------------
  → Γ ⊢ A
```

2.23 Abbreviating de Bruijn indices

```
lookup : Context → ℕ → Type
lookup (A :: Γ) zero    = A
lookup (_ :: Γ) (suc n) = lookup Γ n
lookup [] _             = ⊥-elim impossible
  where postulate impossible : ⊥

count : ∀ {Γ} → (n : ℕ) → Γ ∋ lookup Γ n
count {_ :: Γ} zero    = Z
count {_ :: Γ} (suc n) = S (count n)
count {[]} _           = ⊥-elim impossible
  where postulate impossible : ⊥

#_ : ∀ {Γ} → (n : ℕ) → Γ ⊢ lookup Γ n
# n = ` count n
```

2.24 Renaming

```
Renaming : Context → Context → Set
Renaming Γ Δ = ∀ {C} → Γ ∋ C → Δ ∋ C
```

```
Rebasing : Context → Context → Set
Rebasing Γ Δ = ∀ {C} → Γ ⊢ C → Δ ⊢ C

ext : ∀ {Γ Δ A}
        → Renaming Γ Δ
          -----------------------------------
        → Renaming (A :: Γ) (A :: Δ)
ext ρ z = z
ext ρ (s x) = s (ρ x)

rename : ∀ {Γ Δ}
        → Renaming Γ Δ
          ---------------------------
        → Rebasing Γ Δ
rename-env : ∀ {Γ Γ′ Δ}
  → Renaming Γ Γ′
  → Env Δ Γ
    -------------
  → Env Δ Γ′

rename ρ (‘ x) = ‘ ρ x
rename ρ (L · M) = rename ρ L · rename ρ M
rename ρ ⟨⟨ N , E ⟩⟩ = ⟨⟨ N , rename-env ρ E ⟩⟩
rename ρ ‘zero = ‘zero
rename ρ (‘suc N) = ‘suc rename ρ N
rename ρ (case L M N) = case (rename ρ L) (rename ρ M) (rename (ext ρ) N)
-- rename ρ ⟨⟩ = ⟨⟩
-- rename ρ ⟨ M , N ⟩ = ⟨ rename ρ M , rename ρ N ⟩
rename-env ρ [] = []
rename-env ρ (M :: E) = rename ρ M :: rename-env ρ E

weaken : ∀ {Γ A} → Renaming Γ (A :: Γ)
weaken z = s z
weaken (s x) = s (weaken x)
```

2.25 Simultaneous Substitution

```
Substitution : Context → Context → Set
Substitution Γ Δ = ∀ {C} → Γ ∋ C → Δ ⊢ C

exts : ∀ {Γ Δ A}
    → Substitution Γ Δ
      ----------------------------
```

```
      → Substitution (A :: Γ) (A :: Δ)
exts σ z = ‘ z
exts σ (s x) = rename s_ (σ x)

subst : ∀ {Γ Δ}
      → Substitution Γ Δ
        ----------------
      → Rebasing Γ Δ
subst-env : ∀ {Γ Γ′ Δ}
   → Substitution Γ Γ′
   → Env Δ Γ
     -------------
   → Env Δ Γ′

subst σ (‘ x) = σ x
subst σ (L · M) = subst σ L · subst σ M
subst σ ⟨⟨ N , E ⟩⟩ = ⟨⟨ N , subst-env σ E ⟩⟩
subst σ ‘zero = ‘zero
subst σ (‘suc N) = ‘suc subst σ N
subst σ (case L M N) = case (subst σ L) (subst σ M) (subst (exts σ) N)
subst-env σ [] = []
subst-env σ (M :: E) = subst σ M :: subst-env σ E
```

2.26 Single substitution

```
_[_] : ∀ {Γ A B}
  → A :: Γ ⊢ B
  → Γ ⊢ A
    ------------
  → Γ ⊢ B
_[_] {Γ} {A} N V = subst {A :: Γ} {Γ} σ N
  where
  σ : ∀ {B} → A :: Γ ∋ B → Γ ⊢ B
  σ z     = V
  σ (s x) = ‘ x
```

2.27 Values

```
infix 4 V-⟨⟨_,_⟩⟩
data Value : ∀ {Γ A} → Γ ⊢ A → Set where
```

```
-- functions

V-⟨⟨_,_⟩⟩ : ∀ {Γ Δ A B}
  → (N : A ∷ A ⇒ B ∷ Δ ⊢ B)
  → (E : Env Δ Γ)
    ---------------------
  → Value (⟨⟨ N , E ⟩⟩)

-- naturals

V-zero : ∀ {Γ} →
    -----------------
    Value (`zero {Γ})

V-suc_ : ∀ {Γ} {V : Γ ⊢ `ℕ}
  → Value V
    --------------
  → Value (`suc V)
```

2.28 Helper functions for reduction

```
Env→σ : ∀ {Γ Δ}
  → Env Δ Γ
    -----------------
  → Substitution Δ Γ
Env→σ [] ()
Env→σ (M ∷ E) Z = M
Env→σ (M ∷ E) (S x) = Env→σ E x

make-σ : ∀ {Γ Δ A B} --
  → Env Δ Γ
  → A ∷ A ⇒ B ∷ Δ ⊢ B
  → Γ ⊢ A
    ------------------------------
  → Substitution (A ∷ A ⇒ B ∷ Δ) Γ
make-σ E F X Z = X
make-σ E F X (S Z) = ⟨⟨ F , E ⟩⟩
make-σ E F X (S S x) = Env→σ E x
```

2.29 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-⟨⟨⟩⟩ : ∀ {Γ Δ A B} {N : A ∷ A ⇒ B ∷ Δ ⊢ B} {E : Env Δ Γ} {V : Γ ⊢ A}
    → Value ⟨⟨ N , E ⟩⟩
    → Value V
      ---------------------
    → ⟨⟨ N , E ⟩⟩ · V —→ subst (make-σ E N V) N

  -- 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 : `ℕ ∷ Γ ⊢ A}
    → L —→ L′
      -------------------------
    → case L M N —→ case L′ M N

  β-zero : ∀ {Γ A} {M : Γ ⊢ A} {N : `ℕ ∷ Γ ⊢ A}
      -------------------
    → case `zero M N —→ M

  β-suc : ∀ {Γ A} {V : Γ ⊢ `ℕ} {M : Γ ⊢ A} {N : `ℕ ∷ Γ ⊢ A}
    → Value V
      ----------------------------
    → case (`suc V) M N —→ N [ V ]
```

2.30 Reflexive and transitive closure

```
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
```

2.31 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 (L · M) with progress L
progress (L · M) | step L—→L′ = step (ξ-·₁ L—→L′)
progress (L · M) | done V-L with progress M
progress (L · M) | done V-L | step M—→M′ = step (ξ-·₂ V-L M—→M′)
progress (.(⟨⟨ N , E ⟩⟩) · M) | done V-NE@(V-⟨⟨ N , E ⟩⟩) | done V-M = step (β-⟨⟨⟩⟩ V-NE V-M)
progress ⟨⟨ N , E ⟩⟩ = done V-⟨⟨ N , E ⟩⟩
progress `zero = done V-zero
progress (`suc N) with progress N
progress (`suc N) | step N—→N′ = step (ξ-suc N—→N′)
progress (`suc N) | done V-N = done (V-suc V-N)
progress (case L M N) with progress L
progress (case L M N) | step L—→L′ = step (ξ-case L—→L′)
progress (case .`zero M N) | done V-zero = step β-zero
progress (case .(`suc _) M N) | done (V-suc V-L) = step (β-suc V-L)
```

2.32 Evaluation

```
data Gas : Set where
  gas : ℕ → Gas

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 : ∀ {A} {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
```

2.33 Examples

```
two : ∀ {Γ} → Γ ⊢ `ℕ
two = `suc `suc `zero

plus : ∀ {Γ} → Γ ⊢ `ℕ ⇒ `ℕ ⇒ `ℕ
plus = ⟨⟨ ⟨⟨ case (# 2) (# 0) (`suc ((# 4) · # 0 · # 1)) , # 0 :: # 1 :: [] ⟩⟩ , [] ⟩⟩

2+2 : [] ⊢ `ℕ
2+2 = plus · two · two
```

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc)
open import Relation.Nullary using (¬_)
open import Data.List using (List ; _::_ ; [])
open import Data.List.Relation.Sublist.Propositional using (_⊆_ ; []⊆_ ; base ; keep ; skip)
open import Data.List.Relation.Sublist.Propositional.Properties using (⊆-refl ; ⊆-trans)
import Data.Product using (Σ; _,_; ∃; Σ-syntax; ∃-syntax)

import PCF as S
import Closure as T
open S using (_,_ ; ∅ ; Z ; S_)
open T using (z ; s_ ; ⟨⟨_,_⟩⟩)
open import SubContext

convert-type : S.Type → T.Type
convert-type S.`ℕ = T.`ℕ
convert-type (A S.⇒ B) = convert-type A T.⇒ convert-type B

convert-context : S.Context → T.Context
convert-context ∅ = []
convert-context (Γ , A) = convert-type A :: convert-context Γ

record _⊩_ (Γ : T.Context) (A : T.Type) : Set where
  constructor ∃[_]_∧_
  field
    Δ : T.Context
    Δ⊆Γ : Δ ⊆ Γ
    N : Δ T.⊢ A

Closure : S.Type → S.Context → Set
Closure A Γ = convert-context Γ ⊩ convert-type A

Var→⊆ : ∀ {Γ A} → Γ S.∋ A → convert-type A :: [] ⊆ convert-context Γ
Var→⊆ {Γ , _} Z = keep ([]⊆ convert-context Γ)
Var→⊆ (S x) = skip (Var→⊆ x)

record AdjustContext {A B Γ Δ} (Δ⊆ABΓ : Δ ⊆ A :: B :: Γ) : Set where
  constructor adjust
  field
    Δ₁ : T.Context
    Δ₁⊆Γ : Δ₁ ⊆ Γ
    Δ⊆ABΔ₁ : Δ ⊆ A :: B :: Δ₁
```

```
adjust-context : ∀ {Γ Δ A B} → (Δ⊆ABΓ : Δ ⊆ A :: B :: Γ) → AdjustContext Δ⊆ABΓ
adjust-context (skip (skip {xs = Δ₁} Δ⊆Γ)) = adjust Δ₁ Δ⊆Γ (skip (skip ⊆-refl))
adjust-context (skip (keep {xs = Δ₁} Δ⊆Γ)) = adjust Δ₁ Δ⊆Γ (skip (keep ⊆-refl))
adjust-context (keep (skip {xs = Δ₁} Δ⊆Γ)) = adjust Δ₁ Δ⊆Γ (keep (skip ⊆-refl))
adjust-context (keep (keep {xs = Δ₁} Δ⊆Γ)) = adjust Δ₁ Δ⊆Γ (keep (keep ⊆-refl))

make-env : (Δ : T.Context) → T.Env Δ Δ
make-env [] = T.[]
make-env (A :: Δ) = (T.‘ z) T.:: T.rename-env T.weaken (make-env Δ)

cc : ∀ {Γ A} → Γ S.⊢ A → Closure A Γ
cc {A = A} (S.‘ x) = ∃[ convert-type A :: [] ] Var→⊆ x ∧ (T.‘ z)
cc (S.λ N) with cc N
cc (S.λ N) | ∃[ Δ ] Δ⊆Γ ∧ N₁ with adjust-context Δ⊆Γ
cc (S.λ N) | ∃[ Δ ] Δ⊆Γ ∧ N₁ | adjust Δ₁ Δ₁⊆Γ Δ⊆ABΔ₁ = ∃[ Δ₁ ] Δ₁⊆Γ ∧ ⟨⟨ T.rename (⊆→ρ Δ⊆A
cc (L S.· M) with cc L | cc M
cc (L S.· M) | ∃[ Δ ] Δ⊆Γ ∧ L′ | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M′ with merge Δ⊆Γ Δ₁⊆Γ
cc (L S.· M) | ∃[ Δ ] Δ⊆Γ ∧ L′ | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M′ | subContextSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ = ∃[ Γ₁
cc {Γ} S.‘zero = ∃[ [] ] []⊆ convert-context Γ ∧ T.‘zero
cc (S.‘suc N) with cc N
cc (S.‘suc N) | ∃[ Δ ] Δ⊆Γ ∧ N₁ = ∃[ Δ ] Δ⊆Γ ∧ (T.‘suc N₁)
cc (S.case L M N) with cc L | cc M | cc N
cc (S.case L M N) | ∃[ Δ ] Δ⊆Γ ∧ L′ | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M′ | ∃[ Δ₂ ] skip Δ₂⊆Γ ∧ N′ with merge₃ Δ⊆Γ
cc (S.case L M N) | ∃[ Δ ] Δ⊆Γ ∧ L′ | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M′ | ∃[ Δ₂ ] skip Δ₂⊆Γ ∧ N′ | subContextSum I
  = ∃[ Γ₁ ] Γ₁⊆Γ ∧ (T.case (T.rename (⊆→ρ Δ⊆Γ₁) L′) (T.rename (⊆→ρ Δ₁⊆Γ₁) M′) (T.rename (⊆→
cc (S.case L M N) | ∃[ Δ ] Δ⊆Γ ∧ L′ | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M′ | ∃[ .(T.‘ℕ :: _) ] keep Δ₂⊆Γ ∧ N′ with merge
cc (S.case L M N) | ∃[ Δ ] Δ⊆Γ ∧ L′ | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M′ | ∃[ .(T.‘ℕ :: _) ] keep Δ₂⊆Γ ∧ N′ | subConte
  = ∃[ Γ₁ ] Γ₁⊆Γ ∧ (T.case (T.rename (⊆→ρ Δ⊆Γ₁) L′) (T.rename (⊆→ρ Δ₁⊆Γ₁) M′) (T.rename (⊆→
```

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat using (ℕ; zero; suc)
open import Relation.Nullary using (¬_)
open import Data.List using (List ; _::_ ; [])
open import Data.List.Relation.Sublist.Propositional using (_⊆_ ; []⊆_ ; base ; keep ; skip)
open import Data.List.Relation.Sublist.Propositional.Properties using (⊆-refl ; ⊆-trans)

open import Closure

⊆→ρ : {Γ Δ : Context} → Γ ⊆ Δ → Renaming Γ Δ
⊆→ρ base ()
⊆→ρ (skip Γ⊆Δ) with ⊆→ρ Γ⊆Δ
... | ρ = λ x → s (ρ x)
⊆→ρ (keep Γ⊆Δ) with ⊆→ρ Γ⊆Δ
... | ρ = λ { z → z ; (s v) → s (ρ v) }

record SubContextSum (Γ Δ Δ₁ : Context) : Set where
  constructor subContextSum
  field
    Γ₁ : Context
    Γ₁⊆Γ : Γ₁ ⊆ Γ
    Δ⊆Γ₁ : Δ ⊆ Γ₁
    Δ₁⊆Γ₁ : Δ₁ ⊆ Γ₁

open SubContextSum

record SubContextSum₃ (Γ Δ Δ₁ Δ₂ : Context) : Set where
  constructor subContextSum
  field
    Γ₁ : Context
    Γ₁⊆Γ : Γ₁ ⊆ Γ
    Δ⊆Γ₁ : Δ ⊆ Γ₁
    Δ₁⊆Γ₁ : Δ₁ ⊆ Γ₁
    Δ₂⊆Γ₁ : Δ₂ ⊆ Γ₁

open SubContextSum₃

merge : ∀ {Γ Δ Δ₁} → Δ ⊆ Γ → Δ₁ ⊆ Γ → SubContextSum Γ Δ Δ₁
merge {[]} {[]} {[]} base base = subContextSum [] base base base
merge {[]} {[]} {σ :: Γ} base ()
merge {[]} {σ :: Γ} ()
merge {σ :: Γ} (skip Δ⊆Γ) (skip Δ₁⊆Γ) with merge Δ⊆Γ Δ₁⊆Γ
... | subContextSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ = subContextSum Γ₁ (skip Γ₁⊆Γ) Δ⊆Γ₁ Δ₁⊆Γ₁
```

```
merge {σ :: Γ} (skip Δ⊆Γ) (keep Δ₁⊆Γ) with merge Δ⊆Γ Δ₁⊆Γ
... | subContextSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ = subContextSum (σ :: Γ₁) (keep Γ₁⊆Γ) (skip Δ⊆Γ₁) (kee
merge {σ :: Γ} (keep Δ⊆Γ) (skip Δ₁⊆Γ) with merge Δ⊆Γ Δ₁⊆Γ
... | subContextSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ = subContextSum (σ :: Γ₁) (keep Γ₁⊆Γ) (keep Δ⊆Γ₁) (sk
merge {σ :: Γ} (keep Δ⊆Γ) (keep Δ₁⊆Γ) with merge Δ⊆Γ Δ₁⊆Γ
... | subContextSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ = subContextSum (σ :: Γ₁) (keep Γ₁⊆Γ) (keep Δ⊆Γ₁) (ke

merge₃ : ∀ {Γ Δ Δ₁ Δ₂} → Δ ⊆ Γ → Δ₁ ⊆ Γ → Δ₂ ⊆ Γ → SubContextSum₃ Γ Δ Δ₁ Δ₂
merge₃ Δ⊆Γ Δ₁⊆Γ Δ₂⊆Γ with merge Δ⊆Γ Δ₁⊆Γ
merge₃ Δ⊆Γ Δ₁⊆Γ Δ₂⊆Γ | subContextSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ with merge Γ₁⊆Γ Δ₂⊆Γ
merge₃ Δ⊆Γ Δ₁⊆Γ Δ₂⊆Γ | subContextSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ | subContextSum Γ₂ Γ₂⊆Γ Γ₁⊆
  = subContextSum Γ₂ Γ₂⊆Γ (⊆-trans Δ⊆Γ₁ Γ₁⊆Γ₂) (⊆-trans Δ₁⊆Γ₁ Γ₁⊆Γ₂) Δ₂⊆Γ₂
```

Of course you may want to use several chapters and much more text than here.