2021 年臺灣國際科學展覽會
優勝作品專輯

作品編號　190035

參展科別　電腦科學與資訊工程

作品名稱　Imperative Programming 程式碼與 Functional Programming 程式碼的等價性與其證明，使用 Agda

得獎獎項　大會獎 三等獎

就讀學校　國立臺灣師範大學附屬高級中學

指導教師　李柏翰

作者姓名　陳立凡

關鍵詞　程式語言、程式證明、函數式程式設計

作者簡介

陳立凡，師大附中，文組，指考生。

對數理跟人文都有興趣，喜歡程式語言理論 (Programming Language Theory)。

對這份報告書有疑問或發現有錯誤或單純想聊天請來信 me@koru.me。

摘要

本研究主要考慮在盡量保留可讀性的情況下，找出將 Imperative Programming 程式碼對應的 Functional Programming 的程式碼並證明。

結果如下：

一、if statement 等價於由 ifte 函數所構成的程式碼，其中函數 ifte 定義在本文內

二、某些 for-loop statement 等價於由 foldl 函數所構成的程式碼

三、某些 for-loop statement 等價於由 map 函數所構成的程式碼

Abstract

This research aims to find equivalence between some Imperative Programming code and Functional Programming code without losing readability.

Results are as follows:

1. “if statement” is equivalent to code constructed by the function “ifte”, where “ifte” is defined in the content.
2. Some “for-loop statement” is equivalent to code constructed by the function “foldl”
3. Some “for-loop statement” is equivalent to code constructed by the function “map”

一、前言

(一)、研究動機

在自學程式語言的過程中，我接觸到了編程範式(Programming paradigm)這一概念。在撰寫程式的時候人們可以使用不同的視角去看待程式碼，進而讓程式碼有著不同的風格。有的視角將程式碼看待成一行行的指令，這種風格稱為指令式程式設計 (Imperative Programming，下簡稱 IP)。有的視角將整個程式看待成一個由一堆小函數組合而成的大函數，這種風格稱為函數式程式設計 (Functional Programming，下簡稱 FP)。

在學習 FP 的過程中，我常常看到有人將 IP 的程式碼對應至 FP 的程式碼，例如將 for-loop 對應至 fold，卻沒看過有人證明過這兩段程式碼真的是等價的。本研究試圖填補這段空缺。

(二)、Functional Programming 簡介

Functional Programming (下簡稱 FP) 是一種程式設計的風格，相對於 Imperative Programming (下簡稱 IP)。IP 將程式看做一行一行的指令，而 FP 則將程式看做由一堆小函數組合而成的大函數。這樣的差距使得 FP 當中沒有 loop ，取而代之的是遞迴函數。詳閱 https://wiki.haskell.org/Functional_programming ，在此不贅述。

(三)、符號使用

為了方便，本研究使用符號的方式會較偏向 Functional Programming 中的使用方式，可能與常見的方式稍有不同，在此說明。

名稱	常見	本研究	說明
函數應用	$f(x)$	$f\ x$	f 是函數，x 是參數
多變數的函數應用	$f(x,y)$	$(f x)\ y$ 可寫為 $f\ x\ y$	f 是一個函數，接收一個參數，回傳「接收一個參數回傳一個值的函數」如此可視為 f 接收兩個參

			數。此稱為 Currying。
函數類型		$f: A \rightarrow B$	f 是函數，A 是參數類型，B 是回傳值的類型
多變數函數類型		$f: A \rightarrow (B \rightarrow C)$ 可寫為 $f: A \rightarrow B \rightarrow C$	一樣為 Currying

(四)、函數介紹

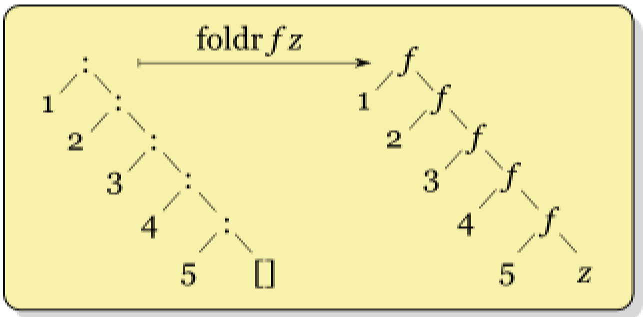
本研究使用了一些在 Functional Programming 中常見的函數，在此說明。

1. fold

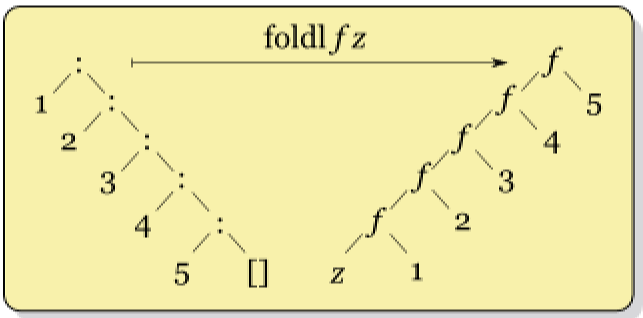
fold 是一個對 List 做處理的函數，fold 有兩種版本，foldr 跟 foldl，以下兩張圖片簡單說明了 [1, 2, 3, 4, 5] 這個 List 在經過 foldr f z 跟 foldl f z 會得到什麼，其中 f 是一個接收兩個參數的函數。

例如 foldr (+) 0 [1, 2, 3] = 6，其中 (+) 代表加法函數

詳細請見 https://wiki.haskell.org/Fold，在此不贅述。



(Right fold transformation，取自 HaskellWiki)



(Left fold transformation，取自 HaskellWiki)

2. map

map 是一個對 List 做處理的函數，類型如下：

$$map: (A \to B) \to ListA \to ListB$$

map f xs 代表對每個 xs 中的值 apply f

例如 map (+4) [1, 2, 3] = [5, 6, 7] ，其中 (+4) 代表 f(x) = x+4 的 f

(五)、Agda 介紹

Agda 是一個依賴類型（dependently typed）的函數式程式語言，亦可基於 Curry–Howard correspondence 作為證明輔助器使用。本研究即用此語言證明。

詳細可參閱：https://wiki.portal.chalmers.se/agda/pmwiki.php

或這個：https://plfa.github.io/ 此書後簡稱 PLFA

依賴類型是一類類型系統，其特色為允許 Type 中有值作為參數，例如 Vec Nat 3 表示長度為 3 ，每個元素皆為 Nat 的向量。

Curry–Howard correspondence 是類型系統和邏輯系統間的同構關係，可用一句話「類型及命題，程式即證明」描述。

例如我們可以先定義自然數：

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

這代表自然數有兩個 constructor，一個是 zero，一個是 suc，後者接收一個自然數並回傳一個自然數。也就是說 zero、suc zero、suc (suc zero) 依此類推都是自然數。

然後我們可以定義關係 _≤_：

```
data _≤_ : ℕ → ℕ → Set where

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

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

(取自 PLFA)

這個 Data Type 有兩種 constructor，z≤n 跟 s≤s ，前者對於任意數字 n，都能夠建構出 zero ≤ n 這個 type 的成員。而後者則接受 m ≤ n 這個 type 的成員，回傳 suc m ≤ suc n 這個 type 的成員，其中 suc x 表示 x 的後繼數。

而透過上述定義，我們可以證明對於任何數 n，n ≤ n，證明如下：

```
≤-refl : ∀ (n : ℕ)
    -----
  → n ≤ n
≤-refl zero = z≤n
≤-refl (suc n) = s≤s (≤-refl n)
```

≤-refl 可被看成是從任意自然數 n 到 n ≤ n 這個 type（命題）的成員（證據）的函數，該函數是遞迴定義的，在參數為 zero 的時候會回傳 z≤n，在參數為 suc n 的時候會回傳 s≤s (≤-refl n)，由於每個自然數 n 都有對應的 ≤-refl n，如此即證明了「對於所有自然數 n，n ≤ n」。也就是說「 ∀ {n : ℕ} → n ≤ n 」這串既是 type，也是命題。而後面既是函數的構造，也是命題的證明。

本研究使用之 Agda 版本為 2.6.0.1

Agda 有 standard-library，本研究中使用了這些，版本為 1.2：

關於 standard-library 可參閱：https://github.com/agda/agda-stdlib

```
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; trans; cong; cong-app)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; _≡⟨_⟩_; _∎)
open import Data.Nat using (ℕ; zero; suc; _+_)
open import Data.Bool using (Bool; true; false)
open import Data.String using (String)
import Data.Vec as V
open V using (Vec; updateAt; _[_]≔_)
open import Relation.Nullary using (yes; no)
open import Data.Empty
import Data.Vec.Properties as VP
import Data.List as L
open L using (List; foldr; foldl; _[_]∷=_)
import Data.List.Properties as LP
open import Data.Fin using (zero; suc; Fin)
open import Function
open import Data.Empty using (⊥; ⊥-elim)
```

另外介紹一些 Agda 的語法，礙於篇幅不能介紹所有本研究用到的語法：

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

這是 Agda 中定義函數的方式，第一行是函數類型，第二三行代表該函數在不同情況下的回傳值，例如當遇到 zero + n，其中 n 是任意自然數，就直接回傳 n，當遇到 (suc m) + n，就回傳 suc (m + n)。

Agda 中 λ 是定義匿名函數的方法，例如 (λ x → x+3) 就代表一個接收 x 回傳 x+3 的函數。

Agda 中的單行註解符號是「--」，該行符號後的文字全都是註解。

Agda 亦有些常用的符號和函數，以下介紹：

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

（取自 PLFA）

這是內建函示庫裡「相等」的定義，透過 refl 可以構造出 x ≡ x 這個 type 的成員（證據）。

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

(取自 PLFA)

這是標準函式庫裡的函數，先接收一個 f 做為參數，接著從 x ≡ y 到 f x ≡ f y。

`begin` `≡⟨ ⟩` `∎`，可以讓證明寫得像是一般數學上的證明一樣，稱為等式推理 (equational reasoning)。

以證明加法結合律為例：

```
+-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)
  ∎
```

(取自 PLFA)

括號內的東西表示上下相等的原因，沒東西代表 Agda 可以自己推出來。最後那個方形代表 Q. E. D.。

這個證明是用數學歸納法寫出來的，分別對 zero 和 suc m （也就是數歸中常見的 n = k+1）兩種情形作討論，在中間 `cong suc (+-assoc m n p)` 的部分用到了假設（在 n = k 時成立）。

二、研究方法與過程

(一)、模型

Imperative Programming 的程式是由一行一行的 statement 所構成，每個 statement 都代表著對狀態的操作、改變。為了方便，這裡僅討論以下條件的 IP 程式碼

1. 整段程式碼必須可以被當成是一個 pure function，即不得與此段程式碼外的事物有互動 (e.g. 讀寫輸出入、讀寫檔案) ，否則難以討論
2. Statement 僅有 variable assignment、if statement、for-loop 三種

我們可以把每個 statement 都想像成是從狀態到狀態的函數，依序應用(apply)到狀態上。而整個程式就是一個 statement。

對於狀態 st 和變數名稱 a，有兩項操作：

$get\ a\ st$ 表示 st 中 a 對應的值

$set\ a\ v\ st$ 表示將 st 中 a 所對應的值改成 v 後新的狀態

首先我們定義 expression，以 e 表示，其語法由以下給出：

$$e \coloneqq v \mid a \mid f(e, e \dots e)$$

其中 v 是任何 value，可能是 boolean、數字或是函數等。a 是變數名稱。f 是函數，而我們目前規定函數只能在 meta language 中被定義。

其求值方式由函數 $E: Expression \times State \rightarrow Value$ 給出：

$$E(v, st) = v$$

$$E(a, st) = get\ a\ st$$

$$E(f(e_1, e_2 \dots e_n), st) = f\ E(e_1, st)\ E(e_2, st) \dots E(e_n, st)$$

而 Statement 的定義如下：

$$S \coloneqq a \leftarrow e$$

$$S \coloneqq if\ e\ then\ s\ else\ s$$

$$S \coloneqq for\ i\ in\ ls\ do\ s$$

$$S \coloneqq for\ i < n\ ls\ do\ s$$

其語義分別為下：

$$[\![a \leftarrow e]\!]\ st = set\ a\ E(e, st)\ st$$

$$[\![if\ true\ then\ s_1\ else\ s_2]\!] = [\![s_1]\!]$$

$$[\![if\ false\ then\ s_1\ else\ s_2]\!] = [\![s_2]\!]$$

$$[\![if\ c\ then\ s_1\ else\ s_2]\!]\ st = [\![if\ E(c, st)\ then\ s_1\ else\ s_2]\!]\ st$$

$$[\![for\ i\ in\ xs\ do\ s]\!]\ st$$

$$= \big(foldr\ \big(\lambda x. f. (f \circ s[i \coloneqq x])\big)\ id\ E(xs, st)\big)\ st$$

其中 id 是 identity function，即 $id(x) = x$。而 $s[i \coloneqq x]$ 的意思是將 s 中出現的 i 替換成 x。

$$[\![for\ i < n\ do\ s]\!]\ st$$

$$= \big(foldr\ \big(\lambda x. f. (f \circ s[i \coloneqq x])\big)\ id\ [0..(n-1)]\big)\ st$$

其中 [0..(n-1)] 代表 0 到 n-1 的所有自然數。

(二)、模型實作

為了證明，本研究透過 Agda 實作了上述模型

本研究定義了以下幾種物件：

1. Val (值)

```
data Val : Set where
    nat  : ℕ → Val
    bool : Bool → Val
    str  : String → Val
    arr  : (n : ℕ) → Vec Val n → Val
```

Val 是一種資料類型，有四種可能

(1) nat x ，其中 x 是自然數

(2) bool x ，其中 x 是布林值

(3) str x ，其中 x 是字串

(4) arr n x ，其中 x 是每個元素都是 Val 所構成的長度為 n 的向量

例如 (nat 3) 就是一個 Val

而對這四種可能，分別定義了可以取出內容的函數。

```
getNat : Val → ℕ
getNat (nat x) = x
getNat _ = CRASH

getBool : Val → Bool
getBool (bool x) = x
getBool _ = CRASH

getStr : Val → String
getStr (str x) = x
getStr _ = CRASH

getList : Val → List Val
getList (arr n x) = V.toList x
getList _ = CRASH

getVec : (n : ℕ)  → Val → Vec Val n
getVec n (arr m x) with ( (Data.Nat._≟_) n m)
getVec n (arr .n x) | yes refl = x
...                 | no    p = CRASH
getVec _ _ = CRASH
```

其中 getList 和 getVec 差在一個回傳 List，一個回傳 Vec，後者的 type 中包含長度資訊。

由於本實作未靜態檢查型別，定義了 CRASH 以處理存取到錯誤的變數的情形。其中 CRASH 的定義如下，代表著該種可能不該被執行到。

```
postulate CRASH : ∀ {A : Set} → A
```

例如 testCRASH 就會回傳 CRASH

```
testCRASH : Val
testCRASH = add (str "a") (nat 3)
```

2. State (狀態)

```
State : N → Set
State n =  Vec Val n
```

實作上我們把狀態當成 n 維向量，而變數名稱則以數字表示，代表其在向量中的位置。

State n 是一種資料類型，n 是參數，相當於每個元素都是 Val 所構成的 n 維向量。

```
_▹_ : ∀ {n : N} → State n → Fin n → Val
st ▹ a  = V.lookup st a
```

可用以上的符號來取得變數位置對應的值，也就是模型中的 get 函數，例如 st ▹ a 就是從狀態 st 中取得 a 對應的值。

3. Expr (expression)

```
Expr : N → Set
Expr n = State n → Val
```

為了方便，在實作端並非額外用求值函數 E 求值，而是直接將 expression 設計成從狀

態到值的函數。

Expr n 是一種資料類型，相當於接收 State n 作為參數，回傳 Val 的函數

「將 expression apply 到 state 上，取得實際的值」這個動作稱為「取值」

4. Stmt (statement)

```
Stmt : ℕ → Set
Stmt n = (State n → State n)
```

Stmt n 是一種資料類型，相當於從 State n 到 State n 的函數。

可用以下函數合成，代表先執行 s1 再執行 s2。

```
_▮_ : ∀ {n : ℕ} → Stmt n → Stmt n → Stmt n
s1 ▮ s2 = s2 ∘ s1
```

本研究中使用的 Stmt 共有以下幾種

(1) Variable Assignment

```
_←_ : ∀ {n : ℕ} → Fin n → Expr n → Stmt n
(a ← e) st = st [ a ]≔ (e st)
```

a 是變數位置，e 是 expression，st 是狀態。不難發現這相當於模型中的 $a \leftarrow e$。

右邊的 `st [ a ]:= (e st)` 意思是將 st 中 a 位置的值替換成 e 取值後的結果，也就是模型中 $set\ a\ E(e, st)\ st$。

這種 statement 有一個變體：

```
_⟵_ : ∀ {n : ℕ} → Fin n → Val → Stmt n
(a ⟵ v) st = st [ a ]≔ v
```

圖中箭頭為長箭頭。此 statement 與上面差別在於他直接接收一個 Val，而非需要取值的 Expr n

(2) if then else

```
if-then-else : ∀ {n : ℕ} → Val → Stmt n → Stmt n → Stmt n
if-then-else (bool true) stmt1 stmt2 = stmt1
if-then-else (bool false) stmt1 stmt2 = stmt2
if-then-else _ _ _ = CRASH
```

```
IF_THEN_ELSE  : ∀ {n : ℕ} → Expr n → Stmt n → Stmt n → Stmt n
(IF e THEN stmt1 ELSE stmt2) st = (if-then-else (e st) stmt1 stmt2) st
```

IF e THEN stmt1 ELSE stmt2 會先對 e 取值，並依其是 true 還是 false 判斷要回傳 stmt1 還是 stmt2。不難發現這相當於模型中的 if statement。

(3) for loop

```
for-in-do : ∀ {n : ℕ} → (Expr n) → (Val → Stmt n) → Stmt n
for-in-do {n} xs i-to-stmt st =
    foldr (λ i f → f ∘ (i-to-stmt i) ) id (getList (xs st)) st

for-in-do-syntax = for-in-do

syntax for-in-do-syntax xs (λ i → stmt) = FOR[ i ∈ xs ]DO stmt
```

在模型實作上將 $s[\![i \coloneqq x]\!]$ 改成直接接收一個「接收 i 返回 statement」的函數 i-to-stmt。而底下

`syntax for-in-do-syntax xs (λ i → stmt) = FOR[ i ∈ xs ]DO stmt`

此行代表左邊的 `for-in-do-syntax xs (λ i → stmt)` 可以寫成右邊的 `FOR[ i ∈ xs ]DO stmt` 。

簡單來說就跟一般的 foreach loop 一樣。不難發現跟模型中的 for statement 語法一樣。

例如以下的 test1 = (nat 6) :: []

```
arr1 : Val
arr1 = arr 3 ((nat 1) ∷ ( (nat 2) ∷ ( (nat 3) ∷ []))) where open V

add : Val → Val → Val
add (nat x) (nat y) = nat (x + y)
add _ _ = CRASH

st1 : State 1
st1 = (nat 0) ∷ [] where open V

test1 : State 1
test1 =
    (
        FOR[ i ∈ const arr1 ]DO
            (a ← λ st → add (st ▹ a) i)
    )
    st1
        where
            a = zero
```

另外還有變體（一般的 for-loop）

```
for-do : ∀ {n : ℕ} → (m : ℕ) → (Fin m → Stmt n) → Stmt n
for-do {n} m i-to-stmt =
    foldr (λ i acc → acc ∘ (i-to-stmt i) ) id (L.allFin m)

for-do-syntax = for-do

syntax for-do-syntax m (λ i → stmt) = FOR[ i < m ]DO stmt
```

(三)、等價語法與其證明

1. if else 與 ifte

定義函數 ifte

```
ifte : ∀{A : Set} → Bool → A → A → A
ifte true  t f = t
ifte false t f = f
```

Theorem 1

$$[\![if\ c\ then\ a \leftarrow x\ else\ a \leftarrow y]\!] = [\![a \leftarrow ifte(c, x, y)]\!]$$

也就是說以下兩者是等價的

```
IF e
THEN a ← e1
ELSE a ← e2
```

```
(a ← λ st → (ifte (getBool (b st)) e1 e2) st )
```

證明：

```
thm1 : ∀ {n : ℕ}
       → (a : Fin n)
       → (b : Expr n)
       → (e1 : Expr n)
       → (e2 : Expr n)
       → (st0 : State n)
       → (IF b THEN (a ← e1) ELSE (a ← e2)) st0
               ≡ (a ← λ st → (ifte (getBool (b st)) e1 e2) st ) st0
thm1 a b e1 e2 st0 =
    begin
       (IF b THEN (a ← e1) ELSE (a ← e2)) st0
    ≡⟨⟩
       if-then-else (b st0) (a ← e1) (a ← e2) st0
    ≡⟨ h (b st0) ⟩  -- 因為 h (b st0) 所以上面等於下面，h 定義在下面
       (a ← λ st → (ifte (getBool (b st)) e1 e2) st ) st0
    ∎
```

```
where
    h : ∀ (b´ : Val)
        → (if-then-else b´ (a ← e1) (a ← e2)) st0
                ≡ (a ← ifte (getBool b´) e1 e2 ) st0
        -- 對於所有 b’ ，上面等式成立
    h (bool false) = refl
    h (bool true) = refl
    h _ = CRASH
    -- 針對 b’ 的各種情形作討論，不管是 (bool true) 或 (bool false) ，
    Agda 都能自動確認相等，所以只要寫 refl 就好
    -- 如果 b’ 不屬於上述情形，那就代表程式出錯了，所以填上 CRASH
```

2. for-loop 與 foldl

Theorem 2

$$[\![for\ i\ in\ xs\ do\ a \leftarrow fn(a,i)]\!] = [\![a \leftarrow (foldl\ fn\ a\ xs)]\!]$$

也就是說以下兩者是等價的：

```
FOR[ i ∈ exprXs ]DO
    a ← λ st → (fn (st ▹ a) i )
```

```
a ← λ st → (foldl fn (st ▹ a) (getList (exprXs st)))
```

透過 `foldr-universal` 性質可證明，詳見附錄 IPLang.agda 的 `thm2` (p. 25)

```
foldr-universal : ∀ (h : List A → B) f e → (h [] ≡ e) →
                  (∀ x xs → h (x ∷ xs) ≡ f x (h xs)) →
                  h ≗ foldr f e
foldr-universal h f e base step []       = base
foldr-universal h f e base step (x ∷ xs) = begin
  h (x ∷ xs)          ≡⟨ step x xs ⟩
  f x (h xs)          ≡⟨ cong (f x) (foldr-universal h f e base step xs) ⟩
  f x (foldr f e xs)  ∎
```

3. for-loop 與 map

Theorem 3

$$[\![for\ i < m\ do\ dest \leftarrow fn(xs[i])]\!] = [\![dest \leftarrow (map\ \ fn\ \ xs)]\!]$$

也就是說以下兩行程式碼是等價的：

```
FOR[ i < m ]DO
  dest [ i ]← λ st → (fn (xs [ i ]))
```

```
dest ← λ st → arr m (V.map fn xs)
```

證明請見附錄 IPLang.agda 的 `thm3` (p. 29)

三、結論與未來展望

(一)、結論

1. if then else 可以透過 ifte 函數處理

$$[\![if\ c\ then\ a \leftarrow x\ else\ a \leftarrow y]\!] = [\![a \leftarrow ifte(c, x, y)]\!]$$

2. 某些 for-loop 與 foldl 等價

$$[\![for\ i\ in\ xs\ do\ a \leftarrow fn(a, i)]\!] = [\![a \leftarrow (foldl\ fn\ a\ xs)]\!]$$

3. 某些 for-loop 跟 map 等價

$$[\![for\ i < m\ do\ dest \leftarrow fn(xs[i])]\!] = [\![dest \leftarrow (map\ \ fn\ \ xs)]\!]$$

(二)、未來展望

1. 期望能處理更多語法，例如 while-loop
2. 期望能基於這些等價性，做出自動轉換的程式

四、參考文獻

(一)、引用資料

1. Functional programming. (2020, February 29). HaskellWiki, . Retrieved 22:27, November 1, 2020 from https://wiki.haskell.org/index.php?title=Functional_programming&oldid=63198.

2. Fold. (2019, March 28). HaskellWiki, . Retrieved 17:40, November 1, 2020 from https://wiki.haskell.org/index.php?title=Fold&oldid=62841.

3. agda/agda-stdlib. (2020). Agda Github Community. https://github.com/agda/agda-stdlib (Original work published 2014)

4. The Agda Wiki. (n.d.). Retrieved November 2, 2020, from https://wiki.portal.chalmers.se/agda/pmwiki.php

(二)、參考書目

1. Appel, A. W., & Ginsburg, M. (1998). Modern compiler implementation in C: Basic techniques. Cambridge: Cambridge University Press.

2. Appel, A. W. (1998). SSA is functional programming. ACM SIGPLAN Notices, 33(4), 17-20. doi:10.1145/278283.278285

3. Mitchell, J. C. (2007). Concepts in programming languages. Cambridge: Cambridge University Press.

4. Philip Wadler, Wen Kokke, & Jeremy G. Siek (2020). Programming Language Foundations in Agda.

五、附錄

IPLang.agda

```
module IPFP_Agda.IPLang where
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; sym; trans; cong; cong-app)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; _≡⟨_⟩_; _∎)
open import Data.Nat using (ℕ; zero; suc; _+_)
open import Data.Bool using (Bool; true; false)
open import Data.String using (String)
import Data.Vec as V
open V using (Vec; updateAt; _[_]≔_)
open import Relation.Nullary using (Dec; yes; no)
open import Data.Empty
import Data.Vec.Properties as VP
import Data.List as L
open L using (List; foldr; foldl; _[_]∷=_)
import Data.List.Properties as LP
open import Data.Fin using (zero; suc; Fin)
open import Function
open import Data.Empty using (⊥; ⊥-elim)
import IPFP_Agda.UpdateAtMap
import IPFP_Agda.FoldrPerm

Id : Set
Id = ℕ

postulate impossible : ⊥

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

postulate CRASH : ∀ {A : Set} → A
```

```
data Val : Set where
   nat  : ℕ → Val
   bool : Bool → Val
   str  : String → Val
   arr  : (n : ℕ) → Vec Val n → Val


getNat : Val → ℕ
getNat (nat x) = x
getNat _ = CRASH

getBool : Val → Bool
getBool (bool x) = x
getBool _ = CRASH

getStr : Val → String
getStr (str x) = x
getStr _ = CRASH

getList : Val → List Val
getList (arr n x) = V.toList x
getList _ = CRASH

getVec : (n : ℕ)  → Val → Vec Val n
getVec n (arr m x) with ( (Data.Nat._≟_) n m)
getVec n (arr .n x) | yes refl = x
...                 |  no    p = CRASH
getVec _ _ = CRASH

getVec-cancel : ∀{n : ℕ} → (xs : Vec Val n) → (getVec n (arr n xs)) ≡
xs
getVec-cancel {n }xs with n Data.Nat.≟ n
...                 | yes refl = refl
...                 | no  p = ⊥-elim (p refl)

getVec-cancel′ : ∀{n : ℕ} → (x : Val) → (arr n (getVec n x)) ≡ x
getVec-cancel′ {n} (arr m x) with n Data.Nat.≟ m
getVec-cancel′ {.m} (arr m x) | yes refl = refl
...                           | no  p = CRASH
```

```
getVec-cancel´ _ = CRASH

State : ℕ → Set
State n =  Vec Val n

Stmt : ℕ → Set
Stmt n = (State n → State n)

Expr : ℕ → Set
Expr n = State n → Val

infix 0 _▌_

_▌_ : ∀ {n : ℕ} → Stmt n → Stmt n → Stmt n
s1 ▌ s2 = s2 ∘  s1

infix 1 _←_

_←_ : ∀ {n : ℕ} → Fin n → Expr n → Stmt n
(a ← e) st = st [ a ]≔ (e st)

assign-syntax = _←_

--syntax assign-syntax a (λ st → v) = a ←[ st ]- v

infix 1 _⟵_

_⟵_ : ∀ {n : ℕ} → Fin n → Val → Stmt n
(a ⟵ v) st = st [ a ]≔ v

infix 5 _▹_

_▹_ : ∀ {n : ℕ} → State n → Fin n → Val
st ▹ a  = V.lookup st a

_[_] : ∀ {n : ℕ} → (xs : Vec Val n) → (Fin n) → Val
xs [ i ] = V.lookup xs i

_[_]←_ : ∀ {n m : ℕ}
```

```
          → (xs-name : Fin n)
          → (Fin m)
          → Expr n
          → (st : State n)
          → State n
((_[_]←_) {m = m} xs-name i e) st = (xs-name ⟵ arr m ((getVec m (st ▹
xs-name)) [ i ]≔ (e st)) ) st


if-then-else : ∀ {n : ℕ} → Val → Stmt n → Stmt n → Stmt n
if-then-else (bool true) stmt1 stmt2 = stmt1
if-then-else (bool false) stmt1 stmt2 = stmt2
if-then-else _ _ _ = CRASH

IF_THEN_ELSE  : ∀ {n : ℕ} → Expr n → Stmt n → Stmt n → Stmt n
(IF e THEN stmt1 ELSE stmt2) st = (if-then-else (e st) stmt1 stmt2) st

ifte : ∀{A : Set} → Bool → A → A → A
ifte true  t f = t
ifte false t f = f


for-in-do : ∀ {n : ℕ} → (Expr n) → (Val → Stmt n) → Stmt n
for-in-do {n} xs i-to-stmt st =
   foldr (λ i f → f ∘  (i-to-stmt i) ) id (getList (xs st)) st

for-in-do-syntax = for-in-do

syntax for-in-do-syntax xs (λ i → stmt) = FOR[ i ∈ xs ]DO stmt

for-do : ∀ {n : ℕ} → (m : ℕ) → (Fin m → Stmt n) → Stmt n
for-do {n} m i-to-stmt =
   foldr (λ i acc → acc ∘  (i-to-stmt i) ) id (L.allFin m)

for-do-syntax = for-do

syntax for-do-syntax m ( λ i → stmt) = FOR[ i < m ]DO stmt
```

```
set-get : ∀ {n} → (a : Fin n) → (st0 : State n) →  (a ← λ st → (st ▹
a)) st0 ≡ st0
set-get a st0 = VP.[]≔-lookup st0 a

get-set : ∀ {n} → (a : Fin n) → (v : Val) → (st0 : State n)
          →  ( (( a ⟵ v ) st0) ▹ a) ≡ v
get-set a v st0 = VP.lookup∘update a st0 v



thm1 : ∀ {n : ℕ}
       → (a : Fin n)
       → (b : Expr n)
       → (e1 : Expr n)
       → (e2 : Expr n)
       → (st0 : State n)
       → (IF b THEN (a ← e1) ELSE (a ← e2)) st0 ≡ (a ← λ st → (ifte
(getBool (b st)) e1 e2) st ) st0
thm1 a b e1 e2 st0 =
    begin
       (IF b THEN (a ← e1) ELSE (a ← e2)) st0
    ≡⟨⟩
       if-then-else (b st0) (a ← e1) (a ← e2) st0
    ≡⟨ h (b st0) ⟩
       (a ← λ st → (ifte (getBool (b st)) e1 e2) st ) st0
    ∎

    where
       h : ∀ (b´ : Val)
          → (if-then-else b´ (a ← e1) (a ← e2)) st0 ≡ (a ← ifte
(getBool b´) e1 e2 ) st0
       h (bool false) = refl
       h (bool true) = refl
       h _ = CRASH

thm2´ : ∀ {A : Set} {n : ℕ}
       → (a : Fin n)
       → (fn : Val → A → Val)
       → (xs : List A)
       → (st0 : State n)
```

```
        → (foldr (λ i acc st → acc ( (a ⟵ fn (st ▹ a) i) st)) id xs)
st0 ≡ (a ← λ st → (foldl fn (st ▹ a) xs) ) st0

thm2´ {A} {n} a fn xs st0 =
    begin
        (foldr (λ i acc st → acc ( (a ⟵ fn (st ▹ a) i) st)) id xs) st0
    ≡⟨ refl  ⟩
        (foldr f e xs) st0
    ≡⟨ Eq.cong-app (sym hp) st0  ⟩
        (h xs) st0
    ≡⟨ refl  ⟩
        (a ← λ st → (foldl fn (st ▹ a) xs) ) st0
    ∎
        where
            open L
            h : List A → Stmt n
            h = λ xs´ → (a ← λ st → (foldl fn (st ▹ a) xs´) )
            f = (λ i acc st → acc ( (a ⟵ fn (st ▹ a) i) st))
            e = id
            base : h [] ≡ e
            base = extensionality (set-get a)
            step : (x : A) (ys : List A) → h (x ∷ ys) ≡ f x (h ys)
            step x ys  = extensionality λ st →
                begin
                    h (x ∷ ys) st
                ≡⟨⟩
                    (a ← λ st´ → (foldl fn (st´ ▹ a) (x ∷ ys)) ) st
                ≡⟨⟩
                    (a ⟵ (foldl fn (st ▹ a) (x ∷ ys)) ) st
                ≡⟨⟩
                    (a ⟵ foldl fn (fn (st ▹ a) x) ys) st
                ≡⟨ sym (VP.[]≔-idempotent st a) ⟩
                    (a ⟵ foldl fn (fn (st ▹ a) x) ys) ((a ⟵ fn (st ▹ a)
x) st)
                ≡⟨ Eq.cong-app (cong (λ X →  (a ⟵ foldl fn X ys)) (sym
(get-set a (fn (st ▹ a) x) st))) ((a ⟵ fn (st ▹ a) x) st) ⟩
                    (a ⟵ foldl fn (((a ⟵ fn (st ▹ a) x) st) ▹ a) ys) ((a
⟵ fn (st ▹ a) x) st)
                ≡⟨⟩
```

```
                (h ys) ((a ⟵ fn (st ▹ a) x) st)
            ≡⟨⟩
                f x (h ys) st
            ∎
        hp : h xs ≡ foldr f e xs
        hp = LP.foldr-universal h f e base step xs

thm2 : ∀ {n : ℕ}
       → (a : Fin n)
       → (fn : Val → Val → Val)
       → (exprXs : Expr n)
       → (st0 : State n)
       → (FOR[ i ∈ exprXs ]DO (a ← λ st → (fn (st ▹ a) i ) ) ) st0 ≡
(a ← λ st → (foldl fn (st ▹ a) (getList (exprXs st))) ) st0

thm2 {n} a fn exprXs st0 =
    begin
        (FOR[ i ∈ exprXs ]DO (a ← λ st → (fn (st ▹ a) i)) ) st0
    ≡⟨ refl  ⟩
        (foldr (λ i acc st → acc ( (a ⟵ fn (st ▹ a) i) st)) id xs) st0
    ≡⟨ thm2′ a fn xs st0  ⟩
        (a ← λ st → (foldl fn (st ▹ a) xs) ) st0
    ∎
        where
            open L
            xs = getList (exprXs st0)

thm3′ : ∀ {n m : ℕ}
       → (fn : Val → Val)
       → (xs : Vec Val m)
       → (dest : Vec Val m)
       → (st0 : State n)
       → foldl (λ acc i → arr m ((getVec m acc) [ i ]≔ (fn (xs
[ i ])))) (arr m dest) (L.allFin m)
       ≡ arr m (V.map fn xs)
thm3′ {n} {m} fn xs dest st0 =
    begin
        foldl (λ acc i → arr m ((getVec m acc) [ i ]≔ (fn (xs [ i ]))))
(arr m dest)  (L.allFin m)
```

```
    ≡⟨⟩
        foldl (flip f) (arr m dest)  (L.allFin m)
    ≡⟨ sym (LP.reverse-foldr f dest′ (L.allFin m)) ⟩
        (foldr f dest′ ∘  L.reverse) (L.allFin m)
    ≡⟨⟩
        foldr f dest′ (L.reverse (L.allFin m))
    ≡⟨ hp2 (L.reverse (L.allFin m)) ⟩
        arr m (foldr f′ dest (L.reverse (L.allFin m)))
    ≡⟨ cong (arr m) (sym (IPFP_Agda.FoldrPerm.foldr-reverse (Fin m) f′
dest (L.allFin m) hp3)) ⟩
        arr m (foldr f′ dest (L.allFin m))
    ≡⟨⟩
        arr m (map′ fn xs dest)
    ≡⟨ cong (arr m) (map′≡map fn xs dest) ⟩
        arr m (V.map fn xs)
    ∎
        where
            open IPFP_Agda.UpdateAtMap using (map′; map′≡map)
            f = (λ i acc → arr m ((getVec m acc) [ i ]≔ (fn (xs [ i ]))))
            f′ = (λ i acc → (acc [ i ]≔ (fn (xs [ i ]))))
            dest′ = (arr m dest)
            hp1 = getVec-cancel
            hp2 : (ys : List (Fin m)) → foldr f dest′ ys ≡ arr m (foldr
f′ dest ys)
            hp2 L.[] = refl
            hp2 (y L.∷ ys) =
                begin
                    f y (foldr f dest′ ys)
                ≡⟨ cong (f y) (hp2 ys) ⟩
                    f y (arr m (foldr f′ dest ys))
                ≡⟨⟩
                    arr m ((getVec m (arr m (foldr f′ dest ys))) [ y ]≔
(fn (xs [ y ])))
                ≡⟨ cong (λ X → arr m (X [ y ]≔ (fn (xs [ y ])))) (hp1
(foldr f′ dest ys)) ⟩
                    arr m ((foldr f′ dest ys) [ y ]≔ (fn (xs [ y ])))
                ≡⟨⟩
                    arr m (f′ y (foldr f′ dest ys))
                ∎
```

```
          hp3 : (a b : Fin m) → (c : Vec Val m) → f´ a (f´ b c) ≡
f´ b (f´ a c)
          hp3 a b c =
              begin
                  f´ a (f´ b c)
              ≡⟨⟩
                  (f´ b c) [ a ]≔ (fn (xs [ a ]))
              ≡⟨⟩
                  (c [ b ]≔ (fn (xs [ b ]))) [ a ]≔ (fn (xs [ a ]))
              ≡⟨ hp3-1 (a Data.Fin.≟ b) ⟩
                  f´ b (f´ a c)
              ∎
              where
                  hp3-1 : Dec (a ≡ b)
                          → (c [ b ]≔ (fn (xs [ b ]))) [ a ]≔ (fn (xs
[ a ]))
                          ≡ (c [ a ]≔ (fn (xs [ a ]))) [ b ]≔ (fn (xs
[ b ]))
                  hp3-1 (yes refl) = refl
                  hp3-1 (no  p) = sym (VP.[]≔-commutes c a b p)


thm3 : ∀ {n m : ℕ}
        → (xs : Vec Val m)
        → (dest : Fin n)
        → (fn : Val → Val)
        → (st0 : State n)
        → (FOR[ i < m ]DO
            (dest [ i ]← λ st → (fn (xs [ i ]))  )
          ) st0
        ≡ (dest ← λ st → arr m (V.map fn xs) ) st0

thm3 {n = n} {m = m} xs dest fn st0 =
    begin
        (FOR[ i < m ]DO (dest [ i ]← λ st → (fn (xs [ i ]))  )) st0
    ≡⟨⟩
        (FOR[ i < m ]DO (dest ← λ st → arr m ((getVec m (st ▹ dest))
[ i ]≔ (fn (xs [ i ])))  )) st0
    ≡⟨⟩
```

```
        (FOR[ i < m ]DO (dest ← λ st → g (st ▹ dest) i  )) st0
    ≡⟨⟩
        (foldr (λ i acc st → acc ((dest ← λ st → g (st ▹ dest) i) st) )
id (L.allFin m)) st0
    ≡⟨ thm2´ dest g (L.allFin m) st0 ⟩
        (dest ← λ st → (foldl g (st ▹ dest) (L.allFin m)) ) st0
    ≡⟨⟩
        (dest ⟵ foldl g (st0 ▹ dest) (L.allFin m) ) st0
    ≡⟨ cong-app (cong (λ X → dest ⟵ foldl g X (L.allFin m)) (sym
(getVec-cancel´ (st0 ▹ dest)))) st0 ⟩
        (dest ⟵ foldl g (arr m dest´) (L.allFin m) ) st0
    ≡⟨ cong-app (cong (dest ⟵_) (thm3´ fn xs dest´ st0)) st0 ⟩
        (dest ⟵ arr m (V.map fn xs) ) st0
    ≡⟨⟩
        (dest ← λ st → arr m (V.map fn xs) ) st0
    ∎
        where
            open V using (_∷_; [])
            dest´ = getVec m (st0 ▹ dest)
            g : Val → Fin m → Val
            g acc i = arr m ((getVec m acc) [ i ]≔ (fn (xs [ i ])))
```

UpdateAtMap.agda

```
module IPFP_Agda.UpdateAtMap where

import Relation.Binary.PropositionalEquality as PE
open PE using (_≡_; refl; sym; trans; cong)
open PE.≡-Reasoning using (begin_; _≡⟨⟩_; _≡⟨_⟩_; _∎)

open import Data.Nat as ℕ
open import Data.Fin hiding (_+_)
open import Data.Vec
open import Data.Vec.Properties
open import Function
import Data.List as L
open L using (List)
import Data.List.Properties as LP

foldr-compose : ∀ {A B C : Set}
              → (f : A → B → B)
              → (g : C → A)
              → (z : B)
              → (xs : List C)
              → L.foldr (f ∘ g) z xs ≡ L.foldr f z (L.map g xs)
foldr-compose f g z List.[] = refl
foldr-compose f g z (x L.∷ xs) =
    begin
        L.foldr (f ∘ g) z (x L.∷ xs)
    ≡⟨ refl ⟩
        f (g x) (L.foldr (f ∘ g) z xs)
    ≡⟨ cong (f (g x)) (foldr-compose f g z xs) ⟩
        f (g x) (L.foldr f z (L.map g xs))
    ≡⟨ refl ⟩
        L.foldr f z (L.map g (x L.∷ xs))
    ∎

mapSome : ∀  {A B : Set} {n} → List (Fin n) → (A → B) → Vec A n → Vec
B n → Vec B n
mapSome {A} {B} {n} some-i f xs dest = L.foldr (λ i acc → acc [ i ]≔ (f
(lookup xs i))) dest some-i
```

```
mapSome-ignoreFst : ∀  {A B : Set} {n } → List (Fin n) → (A → B) → Vec
A (suc n) → Vec B (suc n) → Vec B (suc n)
mapSome-ignoreFst {A} {B} {n} some-i f xs dest = L.foldr ( (λ i acc →
acc [ i ]≔ (f (lookup xs i))) ∘ suc ) dest some-i

mapSome-ignoreFst≡d∷mapSome : ∀ {A B : Set} {n : ℕ}
                              → (some-i : List (Fin n))
                              → (f : A → B)
                              → (xs : Vec A n)
                              → (x : A)
                              → (ds : Vec B n)
                              → (d : B)
                              → mapSome-ignoreFst some-i f (x ∷ xs) (d ∷
ds) ≡ d ∷ (mapSome some-i f xs ds)
mapSome-ignoreFst≡d∷mapSome L.[] f xs y ds d = refl
mapSome-ignoreFst≡d∷mapSome {A} {B} {n} (i L.∷ some-i) f xs x ds d =
    begin
        mapSome-ignoreFst (i L.∷ some-i) f (x ∷ xs) (d ∷ ds)
    ≡⟨ refl ⟩
        updateAt
            (suc i)
            ( λ _ → f (lookup xs i))
            (L.foldr
                ( λ i´ acc → updateAt (suc i´) ( λ _ → f (lookup xs
i´)) acc)
                (d ∷ ds)
                (some-i)
            )
    ≡⟨ refl ⟩
        updateAt (suc i) (λ _ → f (lookup xs i)) (mapSome-ignoreFst some-
i f (x ∷ xs) (d ∷ ds))
    ≡⟨ cong (updateAt (suc i) (λ _ → f (lookup xs i))) (mapSome-
ignoreFst≡d∷mapSome some-i f xs x ds d) ⟩
        updateAt (suc i) (λ _ → f (lookup xs i)) (d ∷ (mapSome some-i f
xs ds))
    ≡⟨ refl ⟩
        d ∷ mapSome (i L.∷ some-i) f xs ds
    ∎
```

```
map′ : ∀  {A B : Set} {n} → (A → B) → Vec A n → Vec B n → Vec B n
map′ {A} {B} {n} = mapSome (L.allFin n)

map′-ignoreFst : ∀  {A B : Set} {n} → (A → B) → Vec A (suc n) → Vec B
(suc n) → Vec B (suc n)
map′-ignoreFst {A} {B} {n} f xs dest = L.foldr ( (λ i acc → acc [ i ]≔
(f (lookup xs i))) ∘ suc ) dest (L.allFin n)

map′-ignoreFst≡d∷map′ : ∀ {A B : Set} {n m : ℕ}
                               → (f : A → B)
                               → (xs : Vec A n)
                               → (x : A)
                               → (ds : Vec B n)
                               → (d : B)
                               → map′-ignoreFst f (x ∷ xs) (d ∷ ds) ≡ d
∷ (map′ f xs ds)
map′-ignoreFst≡d∷map′ {n = n} = mapSome-ignoreFst≡d∷mapSome (L.allFin
n)

map′≡map : ∀ {A B : Set} {n : ℕ}
               → (f : A → B)
               → (xs : Vec A n)
               → (dest : Vec B n)
               → map′ f xs dest ≡ map f xs
map′≡map {A} {B} {.0} f [] [] = refl
map′≡map {A} {B} {(suc n)} f (x ∷ xs) (d ∷ ds) =
    begin
        map′ f (x ∷ xs) (d ∷ ds)
    ≡⟨ refl ⟩
        updateAt
            zero
            (λ _ → f x)
            (L.foldr
                (λ i acc → updateAt i (λ _ → f (lookup (x ∷ xs) i)) acc)
                (d ∷ ds)
                (L.tabulate suc)
            )
```

```
    ≡⟨ cong
        ( λ X →
        updateAt
            zero
            ( λ _ → f x)
            (L.foldr
                (λ i acc → updateAt i (λ _ → f (lookup (x ∷ xs) i)) acc)
                (d ∷ ds)
                X
            ))
        (sym (LP.map-tabulate id suc))
    ⟩
        updateAt
            zero
            ( λ _ → f x)
            (L.foldr
                (λ i acc → updateAt i (λ _ → f (lookup (x ∷ xs) i)) acc)
                (d ∷ ds)
                (L.map suc (L.allFin n))
            )
    ≡⟨ cong (λ X → updateAt zero (λ _ → f x) X) (sym (foldr-compose (λ
i acc → updateAt i (λ _ → f (lookup (x ∷ xs) i)) acc) suc (d ∷ ds)
(L.allFin n) )) ⟩
        updateAt
            zero
            ( λ _ → f x)
            (L.foldr
                ( (λ i acc → updateAt i (λ _ → f (lookup (x ∷ xs) i))
acc) ∘ suc)
                (d ∷ ds)
                (L.allFin n)
            )
    ≡⟨ refl ⟩
        updateAt zero (λ _ → f x) (map′-ignoreFst f (x ∷ xs) (d ∷ ds))
    ≡⟨ cong (updateAt zero (λ _ → f x)) (map′-ignoreFst≡d∷map′ {m =
n} f xs x ds d) ⟩
        updateAt zero (λ _ → f x) (d ∷ map′ f xs ds)
    ≡⟨ refl ⟩
        f x ∷ (map′ f xs ds)
```

```
≡⟨ cong (_ ∷_) (map´≡map f xs ds) ⟩
    f x ∷ (map f xs)
≡⟨ refl ⟩
    map f (x ∷ xs)
■
```

FoldrPerm.agda

```
open import Data.List
open import Data.List.Properties
open import Relation.Binary.PropositionalEquality hiding (trans)
import Relation.Binary.PropositionalEquality as PE
open PE.≡-Reasoning using (begin_; _≡⟨⟩_; _≡⟨_⟩_; _∎)

module IPFP_Agda.FoldrPerm (A : Set) where

open import Data.List.Relation.Binary.Permutation.Propositional {_} {A}

foldr-perm : ∀ {B : Set} (f : A → B → B) z xs ys
  → (∀ a b c → f a (f b c) ≡ f b (f a c) )
  → xs ↭ ys
  → foldr f z xs ≡ foldr f z ys
foldr-perm f z xs .xs comm refl = refl
foldr-perm f z .(x ∷ _) .(x ∷ _)  comm (prep x perm) = cong (f x) (foldr-
perm f z _ _  comm perm)
foldr-perm f z (x ∷ y ∷ xs) (y ∷ x ∷ ys)  comm (swap x y perm) =
    begin
        f x (f y (foldr f z xs))
    ≡⟨ comm x y (foldr f z xs) ⟩
        f y (f x (foldr f z xs))
    ≡⟨ cong (λ z → f y (f x z)) (foldr-perm f z xs ys comm perm) ⟩
        f y (f x (foldr f z ys))
    ∎
foldr-perm f z xs ys  comm (trans perm perm₁)
  = PE.trans (foldr-perm f z _ _  comm perm) (foldr-perm f z _ _  comm
perm₁)

post : ∀ (x : A) xs ys → xs ↭ ys → xs ∷ʳ x ↭ ys ∷ʳ x
post x xs .xs refl = refl
post x .(x₁ ∷ _) .(x₁ ∷ _) (prep x₁ perm) = prep x₁ (post x _ _ perm)
post x .(x₁ ∷ y ∷ _) .(y ∷ x₁ ∷ _) (swap x₁ y perm) = trans (swap x₁ y
refl) (prep y (prep x₁ (post x _ _ perm)))
post x xs ys (trans perm perm₁) = trans (post _ _ _ perm) (post _ _ _
perm₁)

prep-post : ∀ (x : A) xs ys → xs ↭ ys → x ∷ xs ↭ ys ∷ʳ x
```

```
prep-post x [] .[] refl = refl
prep-post x (x₁ ∷ xs) .(x₁ ∷ xs) refl = trans (swap x x₁ refl) (prep x₁
(prep-post x xs xs refl))
prep-post x .(x₁ ∷ _) .(x₁ ∷ _) (prep x₁ perm) = trans (swap x x₁ refl)
(prep x₁ (prep-post x _ _ perm))
prep-post x .(x₁ ∷ y ∷ _) .(y ∷ x₁ ∷ _) (swap x₁ y perm) =
   trans (swap x x₁ refl)
   (trans (prep x₁ (swap x y refl))
   (trans (swap x₁ y refl)
   (prep y (prep x₁ (prep-post _ _ _ perm)))))
prep-post x xs ys (trans perm perm₁) = trans (prep-post x _ _ perm) (post
_ _ _ perm₁)

revPerm : (xs : List A) → xs ↭ reverse xs
revPerm [] = refl
revPerm (x ∷ xs) rewrite unfold-reverse x xs = prep-post x xs (reverse
xs) (revPerm xs)

foldr-reverse : ∀ {B : Set} (f : A → B → B) z xs
              → (∀ a b c → f a (f b c) ≡ f b (f a c) )
              → foldr f z xs ≡ foldr f z (reverse xs)
foldr-reverse f z xs comm = foldr-perm f z xs (reverse xs) comm (revPerm
xs)
```

【評語】190035

本研究主題清楚且聚焦，為資訊科學重要的基礎研究。建議在科學驗證與報告上可有更清楚的脈絡。