函數程設與推論
Functional Program Construction and Reasoning

Shin-Cheng Mu

October 23, 2023

目錄

CHAPTER 0

符號、演算、與抽象化

我們先從一個故事開始。

一個島上住著兩種人，騎士 (knight) 與惡棍 (knave) 。騎士總說實話，惡棍總說謊話 — 所謂「謊話」的意思是實話的相反。從外表看不出一個人是騎士或惡棍。你站在一個山洞前。傳說中，山洞盡頭藏著價值連城的黃金。但也有可能並非如此，洞裡其實是一頭龍，一進去就會被吃了。洞口站著一位老先生，看不出是騎士或是惡棍。你該如何設計一個問題問他，得知山洞裡到底有沒有金子呢？

我們不妨先做個熱身題。

例 0.1. 假設在你面前站著兩位居民，A 與 B. 居民 A 說：「如果你問 B 他自己是不是騎士，他會說是。」由此你可知道 A 是騎士或是惡棍嗎？ B 呢？

在讀下去之前，請先嘗試解解看。

0.1 騎士與惡棍之島

剛剛你是怎麼解這問題的？許多人解這類問題用的是窮舉法：假設 A是騎士、B也是騎士，看看會不會有矛盾，接著假設 A是騎士、B是惡棍，看看是否有矛盾... 依此類推。

我們想介紹另一種做法。我們將「A是騎士」簡寫成 A. 如果 A說了某個陳述 S, 我們無法知道 S是否成立，但我們知道

$$\mathrm{A} \equiv \mathrm{S} \ .$$

其中的 $(\equiv)$ 可以理解為邏輯中「若且唯若」的關係，也可以簡單理解為（真假值的）「等於」。若用口語說，當我們寫 $\mathrm{P} \equiv \mathrm{Q}$, 表示 P 和 Q 的真假值相同。確

實，假定 A說了陳述 S，如果 A是騎士，S便是真的；如果 A不是騎士，S便是假的。因此「A說了陳述 S」可寫成 $A \equiv S$。

「若且唯若」這符號有幾個性質。首先，對任何陳述 P, 可知 $P \equiv P$ 必定為真：

$$(P \equiv P) \equiv \mathit{true} \ .$$

我們將這條規則命名為「自反恆真」。[1] 其次，既然 $(\equiv)$ 可以理解為「等於」，理所當然該有交換律：[2]

$$\begin{aligned} & P \equiv Q \\ = \ & Q \equiv P \ . \end{aligned}$$

最後，很少人注意到，真假值的「等於」是有結合律的：對任何 P, Q, 和 R, 我們有這樣的性質：[3]

$$\begin{aligned} & (P \equiv Q) \equiv R \\ = \ & P \equiv (Q \equiv R) \ . \end{aligned}$$

為了練習，我們多考慮一些情況。假設我們問 A：「你是騎士嗎？」而 A 回答「是的！」這可以記成 $A \equiv A$. 然而，根據自反恆真，$A \equiv A$可以化簡為 *true*. 這意味著：在這島上不管你問誰「你是騎士嗎」，對方都會回答「是」。

如果 A說：「B是騎士」呢？這可以記成 $A \equiv B$. 由此我們無法知道 B到底是騎士還是惡棍，但我們可以知道 A和 B是同一種人。

現在想想問題 0.1：居民 A 說「如果你問 B 他自己是不是騎士，他會說是。」這可以記成 $A \equiv (B \equiv B)$. 我們來演算看看：

$$\begin{aligned} & A \equiv (B \equiv B) \\ = \ & \quad \{\text{自反恆真}\} \\ & A \equiv \mathit{true} \\ = \ & \quad \{\text{結合律與自反恆真}\} \\ & A \ . \end{aligned}$$

也就是說，我們不知道 B到底是騎士還是惡棍，卻可知道 A一定是騎士！這段演算中，把 $A \equiv \mathit{true}$ 換成 A 的最後一步，直覺上似乎很直觀。如果要探究理由，其實用了結合律與自反恆真：$(A \equiv A) \equiv \mathit{true}$ 根據結合律可代換為 $A \equiv (A \equiv \mathit{true})$, 因此 $A \equiv \mathit{true}$ 可代換成 A.

再考慮一個熱身題：如果 A說「我和 B是同一種人」呢？這可記成 $A \equiv (A \equiv B)$。我們算算看:

---

[1] 給定二元關係 $\lhd$，如果 $x \lhd x$ 對任何 x 均成立，我們說 $\lhd$ 有自反性 (*reflexivity*). 「自反恆真」即「自反性恆為真」的意思。

[2] $(\equiv)$ 和 $(=)$ 都是「等於」。我們在此處使用不同符號，僅是為了區分出先後次序：同一行內部的相等用 $(\equiv)$, 行與行之間的相等用 $(=)$. 本書其他地方的用法可能不同。

[3] $(P \equiv Q) \equiv R$ 只在 P, Q, 和 R 都是真假值（或邏輯陳述）時有意義，否則 P, Q, 和 R無法用 $(\equiv)$ 串起。反例：$(3 = 3) = 3$ 可化簡成 $\mathit{true} = 3$, 但後式中等號兩邊的值型別不同，不是合法的句子。也許因為我們較常考慮數字的相等，而較少用符號表達真假值的相等，$(\equiv)$ 的結合律並不廣為人知。

$$
\begin{aligned}
& A \equiv (A \equiv B) \\
= & \quad \{ \text{結合律} \} \\
& (A \equiv A) \equiv B \\
= & \quad \{ \text{自反恆真} \} \\
& \mathit{true} \equiv B \\
= & \quad \{ \text{交換律與自反恆真} \} \\
& B \ .
\end{aligned}
$$

這次，我們不知 A是騎士還是惡棍，卻可知 B一定是騎士。

最後，我們該回到本章開頭的問題了。你站在山洞口，面對洞口的老人。姑且稱呼他為 A. 要怎麼想出一個問題問他，用來判斷島上到底有沒有金子呢？之前介紹邏輯符號的目的是讓我們可避免瞎猜，而用代數解未知數的方式把問題推演出來。把「島上有金子」這個命題記為 G. 我們希望問某個問題 Q。如果 A對問題 Q回答「是」，就表示島上有金子，反之則否。

- 「A對問題 Q回答『是』」記為 $A \equiv Q$；
- 「A對問題 Q回答『是』，島上便有金子」其實應該是一個若且唯若的命題，寫成 $G \equiv (A \equiv Q)$。

而根據交換律和結合律，我們可以推演：

$$
\begin{aligned}
& G \equiv (A \equiv Q) \\
= & \quad \{ \text{結合律} \} \\
& (G \equiv A) \equiv Q \\
= & \quad \{ \text{交換律} \} \\
& Q \equiv (G \equiv A) \ .
\end{aligned}
$$

我們可得知 $Q \equiv (G \equiv A)$，也就是說我們要問 A的問題 Q就是 $G \equiv A$：「請問，『島上有金子』和『你是騎士』是不是等價的呢？」[4]

希望這個島上不論騎士或惡棍，邏輯都蠻不錯，才聽得懂這種問題囉！

0.2 讓符號為你工作

回顧看看，我們方才解各種「騎士與惡棍」問題的方法都分為兩個步驟：第一步先把情況以一組符號描述出來，第二步則是依照這些符號的規則下去推演。理想上，第二步比第一步容易，因為我們在進行推演時已不用回頭去思考這些符號的意思，只需照著符號本身的規則不加思索地推演。這麼的好處是能減輕我們思考的負擔。

探究怎樣的符號組成是合法的、一組合法的符號能否經由給定的規則轉換成另一組符號，是語法 (*syntax*)層次上的問題。問這些符號的「意思」是什麼，則是語意 (*semantics*)的層次。以騎士與惡棍問題為例，如果我們回到第一原則去窮舉「如果 A是騎士，B也是騎士.. 如果 A是惡棍，B也是惡棍.. 」等等種種可能，這是在語意上思考。至少對這個問題來說，語意上的解法不僅較繁瑣，也只能用來回答「某人是騎士還是惡棍」類型的問題，而難以用來推演出「該如何問老人，才能得知山洞中是否有黃金？」。上一節的解法則是利用語法幫助

[4] 不妨試試看能否把這句子講得不那麼繞口？

我們：將問題寫下，利用交換律、結合律等等規則時，我們其實沒有回頭去思考「等等，這個式子是什麼意思？」但在符號的幫助之下，我們以更簡潔的方式解了許多個問題。

再舉一個「以符號思考」的例子：你會如何算 28×18? 如果你的思路是「28乘以 19是 10個 28加上 8個 28，而 10個 28是 280，8個 28是... 10個 28減掉 2個 28!」那麼這種思考方式比較接近語意上的。你不斷在思考這個式子「代表什麼」，並試圖尋找較簡單的捷徑。如果你拿起紙筆，甚至在心中畫出了這樣的符號：

$$\begin{array}{r} 28 \\ \times\ 17 \\ \hline \end{array}$$

然後開始照背誦的九九乘法表去推演，此時你用的是語法式的解法。你也許不會思考許多，只以熟悉的規則推著演算的進行。以乘法而言，這兩種作法各有用處。但不可否認地，這套語法式解法的存在使我們可安心地把乘法當作已經解決的問題：我們可以不用思考地作乘法。

許多人見到數學符號就覺得難。事實上，數學符號是發明來簡化問題的。符號與符號之間的規則讓我們可僅用語法、不需在語意上思考，我們因而可把腦力用在更難的問題上。大家看到數學符號就覺得難，其實正是因為它們常被用在不靠數學符號就難以解決的難題上。難的是這些問題，而符號給了我們解決它們的能力。

這一切和程式語言有什麼關係呢？因為，一套好的程式語言就是一套設計良好的符號。它能幫助我們描述問題，並找出解決問題的方法。一些程式語言學者提出的口號「讓符號為你工作 (let the symbols do the work!)」恰好地描述了這門學問的精神。

0.3 抽象化

如前所述，我們解決問題的第一步是以符號將它表達出來。這一步稱作「抽象化 (abstraction)」，通常是較難的一步。

例 0.2. *Mary* 有的蘋果數目是 *John* 的兩倍。*Mary* 發現她的蘋果中有一半已經壞掉了，於是丟了它們。*John* 則吃了一顆蘋果。現在 *Mary* 有的蘋果數目仍是 *John* 的兩倍。請問他們最初各有多少顆蘋果?

你會如何解這問題呢？如果回到第一原則，我們也許可以一步步嘗試：試試看 John 最初有一顆蘋果，Mary 有兩顆的情況是不是合理解答，然後試試看 John 有兩顆，Mary 有四顆的情況... 這是在語意上解問題。就如同我們解騎士與惡棍問題時用窮舉法一樣。

但許多人也許會用代數：把 Mary 最初的蘋果數目用 m表示，John 的蘋果數目用 j表示，寫成這樣的式子：

$$\begin{aligned} m &= 2j \ , \\ m/2 &= 2(j-1) \ . \end{aligned}$$

接下來，用高中程度的代數方法，就可以找出 m和 j的值了。

代數方法是純粹基於語法的技術：我們觀察式子的結構，決定該怎麼作（例如把第二個式子乘以二，兩式相減；或著把 m代換成 $2j$），而不需記得 m和 j分別是什麼意思。

不需依賴「意義」在此是個重要的優點：這表示這套代數方法和特定問題的意義無關，可應用在許許多多場合。可用來解這個問題，也可用來解雞兔問題... 只要能把問題表示成代數式子，就有一個機械化、不需費神思考的解法。

從「Mary 有的蘋果數目是 John 的兩倍... 」到 $m = 2j; m/2 = 2(j-1)$的轉變是一個「抽象化」。抽象化在此的意義是**將不重要的資訊拋棄，只『抽取出』此問題中最關鍵的元素**。由於此處我們的目的是找出「Mary 與 John 最初各有多少顆蘋果」，我們可猜想關於數字的資訊 (例如 $m = 2j$) 是重要的，並猜想其他的一些資訊（蘋果是壞掉還是被吃掉了？兩人丟/吃蘋果的先後順序？）可能是不重要的。因此我們決定只留下關於數字的資訊，並幸運地確實靠此解出了問題。

電腦是個只會處理符號的機器。實體世界的問題之所以能用電腦來解決，仰賴的就是「抽象化」這一步。有人說，整個計算科學就是關於抽象化的學問。

日常用語中，當我們說某事物「很抽象」，通常意味該事物模糊而難以理解。在計算科學中，「抽象」的事物才是最關鍵、最確實、最該把握的。我們前面才說過「數學符號讓事情變容易」，現在又說「抽象過的事物是最確實的」。看來學程式語言久了，讓我們越來越難與常人溝通。最後只好離群索居，躲到騎士與惡棍之島上捉弄來訪的遊客囉！

0.4 抽象化與表達力

在開發較具規模的軟體前，許多軟體方法建議我們分析問題、需求，並寫成形式化的規格。[5] 這是一種抽象化。面對真實的問題時，這一步不容易。

我們再回到蘋果的例子，回想起在該例中，我們選擇抽取出和值有關的資訊，而拋棄了許多其他：因果、先後順序... 等等。

在某些問題或應用中，因果或時間順序是否有可能是重要的？確實有。解這些問題時，我們可能在中途發現選錯了抽象的方式，使得留下的資訊不足以解決該問題。於是我們只好重新開始。有時，我們甚至會發現現有的符號不足以表達這些問題，得設計另一套符號。

這讓我們討論到解決問題重要的*第零步*：在我們能用符號表達問題之前，得先有人設計出一套完善的、足以表達許多問題的符號及其運算規則。為不同目的，我們可能設計不同的符號。一套符號也伴隨著該符號可如何轉變、操作（例如我們見過的交換律、結合律，某些符號碰在一起可化簡、等等）的規則。符號及其規則合起來成為一個「形式 (formal) 系統」。使用設計過的符號與其規則解決問題的研究被稱作「形式方法 (formal method)」。此處的"formal" 一詞，意指我們利用符號的「形式 (form)」來解決問題。[6]

[5] 例如，Abrial [1996] 的 B method 是以一套基於集合論與一階邏輯的形式語言描述軟體規格的方法。

[6] 大學資訊系所的一門必修課"formal language" 中的"formal" 也為「形式」之意。早期台灣將該門課稱為「正規語言」，可能是將 formal 誤解為相對於 casual 的「正式/正規」。

如果我們希望電腦幫我們解問題，需設計的符號就是一套程式語言。這不是一件容易的事：一個程式語言的表達力得強到足以描述所有該語言被設計來解決的問題。一套程式語言是設計者看待世界的抽象觀點。有些程式語言主張世界的狀態可以被一個個指令改變（「把 x 的值更新為 $x+y$; 在螢幕上畫一個方塊... 」）；有些語言認為世界應該看待為一個個物件；有些語言認為世界是許多彼此傳送訊息的共時程序；有些語言認為只要描述出每個個體之前的邏輯關係，交給電腦推論結果，就是很好的計算模型；本書中將介紹的抽象觀點則主張把一切都看成函數：寫程式是定義函數，圖形是座標到顏色的函數，動畫則是時間到圖形的函數...。

晚近的語言設計，尤其是型別系統的設計，採用了許多邏輯學界的結果。本章談「騎士與惡棍之島」時，使用的是命題邏輯 (propositional logic) 的一種分支。一套「邏輯」也可視作一群符號以及其規則。許多人可能不知道，邏輯有許多種。命題邏輯是一套簡單的邏輯，其優點是給定任一合法的句子，都有一套機械性作法可得知其成立與否 — 這個性質稱作「可判定性 (decidability)」。但命題邏輯的*表達力*並不強：許多事情無法在命題邏輯中表達出來。

想表達更細緻的陳述，可使用表達力更強的邏輯。有些邏輯能表達「對所有」、「存在」；有些邏輯能表達先後順序以及「將一直成立」、「將在未來某時間成立」等觀念；有些邏輯可用來表達概念與概念之間的關係。但有得必有失，一套邏輯的表達力只要強到某個程度，就不再有可判定性了，沒有一套固定的方法可知道任意一個邏輯式子的真假。這是理性的限制。在知道形式系統的侷限之下盡力探索並發揮其極限，便是這套學問迷人之處。

因此，設計程式語言時總得掌握這樣的平衡：我們希望語言的表達力強到足以描述我們希望表達的運算程序，但又不希望強到無法掌握其性質。

0.5 正確性

程式語言是設計者選擇用來看待世界的抽象方式。這樣的選擇必然是基於一些考量：如果設計者希望程式能很容易分割、重用，他可能選擇易於將大問題分解成小問題的觀點。如果設計者希望該語言能有效率很高的實作，這語言看待世界的方式可能就和機器的世界觀很接近。

本書選擇的方向則是「*程式語言應能幫助我們確保程式的正確性。*」

但什麼是「正確」？直觀說來，一個程式「正確」的意思是該程式「做到了我們要它做的事。」但，電腦本來不就只能一個指令一個動作地做我們要它做的事嗎？那麼「正確」的意思到底是什麼？

我們考慮一下這個問題：

例 0.3 (最大區段和). *給定一串（有正有負）的數字。請找出一段連續的數字，使其總和越大越好，並傳回這個總和。*

這是經典的「最大區段和」問題。如前面的章節所述，面對一個問題，我們先把它描述成符號。如果 a 是給定的那串數字，N 是其長度，$a[i]$ 是數字中的地 i 個，而 m 是我們想求出的、最大的那個和，最大區段和問題可以描述成這

樣：

$$m = max\{sum(i,j) \mid 0 \leqslant i \leqslant j \leqslant N\} \ , \\ sum(i,j) = \Sigma_{k=i}^{j-1} a[k] \ , \tag{1}$$

其中 $sum\ (i,j)$ 代表 $a\ [i]..a\ [j-1]$ 的和，而 $(\uparrow)\ S$ 找出集合 S 中最大的那個。

但（許多讀者可能也知道）這個問題其實有個線性時間內可執行完畢的解：將這串數字 a 從頭到尾看過一遍，維持兩個變數 s 和 m。每看到一個新數字 $a\ [j]$, 將 s 更新為 0 與 $s+a\ [k]$ 中較大的那個，並把 m 更新為 s 與 m 中較大的那個。寫成程式的話，可能是這樣：

```
s, m = 0, 0
for k in range(0,n-1)
  s = max (0, s + a[k])
  m = max (s, m)
```

那麼，明顯的問題來了。上面的程式和問題描述(1)一點也不像。我們怎麼知道程式真正實作了(1)的要求？

由此談「正確性」，相信清楚多了。數學式(1)給的是一個規格 (specification)，談的是我們要的結果 (*what*), 而程式描述的則是怎麼做 (*how*). 「正確性」總是相對於一個規格而言的：描述「怎麼做」的程式是否真做到了規格的要求？

「最大區段和」的例子可以給我們幾個啟示。首先，把「要什麼」和「怎麼做」牽上關聯，有時是很困難的。當我們說一個程式「很難懂」，其中一個意思是很難看出「為什麼這個程式實作出了這個規格？」（也就是說「為什麼這個程式是正確的？」）看懂一個程式，也就是了解他為何是正確的。

「最大區段和」的例子也告訴我們，「難懂」的程式不一定要很長。上面的程式短短四行，但若沒有輔助解釋，一般人可能很難「看懂」它。[7]

如果規格已經不見了呢？這樣的情形更糟。當我們在沒有規格的情況下問一個程式「是做什麼的」，我們真正問的是「這個程式符合的規格是什麼？」這可能是個難題。回頭看看上面的程式，如果沒有給(1)，你能說得出這個程式在做什麼嗎？

0.6 可演算的程式語言

正確性是如此重要又難以掌握的性質，我們因此希望程式語言能幫助我們確保程式的正確性。我們採用的方法是：給定一個規格，我們希望正確的程式能夠由其規格演算、推導出來。就如同在騎士與惡棍問題中，我們把希望 Q 滿足的性質寫下，然後利用符號的規則算出 Q. 我們希望程式也能如此從其規格被算出來。

以最大區段和問題為例，用我們之後將介紹的符號，我們可以把(1)改寫成：

$$mss = (\uparrow) \cdot map\ sum \cdot segments \ ,$$

[7] 理解這個程式的關鍵是：m永遠是 $a\ [0]..a\ [k]$中所有連續區段最大的和，而 s永遠是 $a\ [0]..a\ [k]$中最右端為 $a\ [k]$的連續區段中最大的和。這兩個條件是該迴圈的恆式 (*loop invariant*)。恆式是了解一個迴圈最重要的資訊。

α	β	γ	δ	ε	ϛ	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο	π	ϙ
1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90

ρ	σ	τ	υ	φ	χ	ψ	ω	ϡ	͵α	͵β	͵γ
100	200	300	400	500	600	700	800	900	1000	2000	3000

Figure 0.1: 希臘數字。

其意思大約是「給定一串數字，找出它的所有連續區段，算每一個區段的和，然後傳回其中最大的。」接下來的問題是，*mss* 有沒有比較快的實作？我們希望可以用代數方法，由 *mss* 的規格出發：

$$
\begin{aligned}
& (\uparrow) \cdot \mathit{map\ sum} \cdot \mathit{segments} \\
= \quad & \{\ \mathit{segments}\ \text{的定義}\ \} \\
& (\uparrow) \cdot \mathit{map\ sum} \cdot \mathit{concat} \cdot \mathit{map\ inits} \cdot \mathit{tails} \\
= \quad & \{\ \text{某定理}\ \} \\
& \ldots \\
= \quad & \{\ \text{另一些定理}\ \} \\
& (\uparrow) \cdot \mathit{scanr}\ (\oplus)\ 0\ .
\end{aligned}
$$

最後的 $(\uparrow) \cdot \mathit{scanr}\ (\oplus)\ 0$ 相當於前一節的程式，只是以不同的符號表示。

我們可以由規格出發，在符號上操作，將程式如同求代數的解一樣地算出來。只要每個小步驟都正確，最後的程式就是正確的。這套技術稱作「程式推導 (program derivation)」或「程式演算 (program calculation)」，將是本書重要的主題。而在本書的觀點中，函數編程相較於其他典範的特色與優勢是函數語言是一套適合演算的符號系統。

適合演算的符號 我們舉個例子，談談符號「適合演算」是怎麼回事。圖 0.1中顯示的是西元前四世紀左右希臘人使用的數字表示法。以希臘字母表為順序，1, 2,...到 9 分別寫成 α, β,...θ. 每遇到 10 的乘冪便換一組符號，如 10, 20, ...寫成 ι, κ...。11, 12, 13 分別寫成 ια, ιβ, ιγ, 21, 22, 23 則寫成 κα, κβ, κγ. 653可寫成 χνγ. 這套表示法的缺點包括不易表示大數，以及不易做演算：我們需要知道 δ 加上 β 等於 ϛ, 以及 μ 加上 κ 等於 ξ。[8]

與之對比的是瑪雅數字。這套數字系統的年代不詳，據估計可能在西元四至世紀採用。一個點代表 1, 一條橫杠代表 5。例如 19 寫成 𝍠. 瑪雅數字採用 20 進位，較大的位數寫在上方。因此

代表 $7(\doteq) \times 20 + 11(\doteq)$, 也就是 151. 另有個例外：第三個數字為 18（而非 20）的乘冪 — 因為 $18 \times 20 = 360$ 接近一年的日數。圖 0.2 中左邊由上至下是以瑪雅數字寫成的 3212199. 值得注意的是，此類以符號的位置表示其量級的數字表

[8] 西元前八世紀，希臘人曾使用以 I, Π, Δ, H 分別表示 1, 5, 10, 100 的表示法。後來被圖 0.1中的系統取代。無論如何，希臘人仍以此發展出了進步的數學。

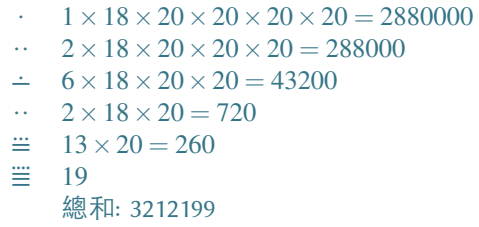
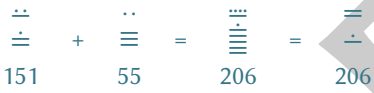


Figure 0.2: 以瑪雅數字表示 3212199.

示法大都需要一個相當於「零」的佔位符號。在瑪雅數字中，該符號寫成  。因此 20 可寫成  , 與只有一個點的 1 區分。[9]

如何用瑪雅數字作加法？以下顯示的是 151 + 55:



我們可先把兩個個位數（∴與 𝌆）和兩個 20 位數（∴與 ··）分別相加。其中，個位數相加的結果是  — 超過了 20！於是我們把其中四個橫槓（即一個 20）往上進位成一個點，和 20 位數的四個點結合成另一個橫槓。

有趣的是，我相信大部分的讀者即使可能是第一次看到這種數系，也從未使用過 20 進位，還是可迅速理解以上的加法，甚至可以舉一反三地自己試試看。這就是一個*適合演算*的符號系統給我們的便利。

對本書來說，函數語言的價值便是 — 它是個適於演算的語言。

0.7 相關資料

本章中的騎士與惡棍問題所使用的邏輯形式稱作*演算邏輯* (calculational logic)，其主要精神是將命題邏輯與述語邏輯表達為適合用於等式推導與演算的形式。Gries and Schneider [2003] 認為大部分形式邏輯系統是為了研究邏輯本身的特性而設計，而演算邏輯則是為了以邏輯解決問題而開發的。演算邏輯由程式語言學者們於 80 年代初期漸漸發展出來。[10] Dijkstra and Scholten [1990] 將演算邏輯用於指令式程式語言的語意中。本章中的騎士與惡棍問題由 Backhouse [2003] 中節錄而來。這是一本介紹以形式方式構思演算法以解決問題的教科書。本章只用到了 (≡) 一個運算子，該書中則有更完整的介紹。Gries and Schneider [October 22, 1993] 則是一本以演算邏輯為基礎的離散數學教科書。

「讓符號為你工作」這句口號可在 Dijkstra 的許多著作中見到（例如 Dijkstra and van Gasteren [1986] , Dijkstra [2000]）。Dijkstra [2004] 寫道：「設計程式的

[9] 一些早期數字表示法中有此種佔位符號，但並不見得把「零」視為可獨立存在的一個數。「將零視為一個數字」是數學史上的重要發明，許多數學性質都需有它的存在才得以描述。程式語言中也有類似的情況：有一個「不做任何事的指令」或一個「將輸入照樣傳回的函數 (即 *id*)」對於描述程式的性質是很重要的。

[10] Gries and Schneider [2003] 將此歸功於 Roland Backhouse, Edsger W. Dijkstra, Wim H.J. Feijen, David Gries, Carel S. Scholten, 以及 Netty van Gasteren 等人。他們的共同研究主題是以形式方法開發程式。

人最好把程式當作精巧的算式。而我們知道只有一種設計精巧算式的可靠方法：用符號操作來推導它。我們得讓符號為我們工作，因為這是已知唯一在大尺度下仍可行的方法。」Misra 則在 1988 年 Marktoberdorf 暑期課程的演講 [Misra, 1989] 中虛構了一個故事：使用羅馬數字的羅馬人被推銷比較易於演算的印度-阿拉伯數字系統，卻不以為然地說「我們已經有奴隸為我們工作了 !」

希臘與瑪雅數字的例子取自 Mazur [2014]。該書對於種種符號的演進有更詳盡的介紹。

CHAPTER 1

值、函數、與定義

語言是概念的載體。如第 0章所述，程式語言不僅用來表達概念，也用於演算，分擔我們思考的負擔。在本書中，為便於精確說明，我們也不得不選一個語言。

本書中使用的語言大致上基於 Haskell, 但為適合本書使用而經過簡化、修改。我們將在本章初步介紹 Haskell 語言的一小部分，包括在 Haskell 中何謂「計算」、值與函數的定義、常見的資料結構，以及串列上的常用函數。目的是讓讀者具備足夠的基本概念，以便進入之後的章節。也因此，本書中所介紹的語言並非完整的 Haskell, 本書也不應視作 Haskell 語言的教材。對於有此需求的讀者，我將在本章結尾推薦一些適合的教科書。

1.1 值與求值

Haskell 是個可被編譯 (compile)、也可被直譯 (interpret) 的語言。Haskell 直譯器延續了 LISP 系列語言的傳統，是個「讀、算、印 (read, evaluate, pring)」的迴圈 — 電腦從使用者端讀取一個算式，算出結果，把結果印出，然後再等使用者輸入下一個算式。一段與 Haskell 直譯器的對話可能是這樣：

```
Main > 3 + 4
7
Main > sum [1..100]
5050
Main >
```

在此例中，Main> 是 Haskell 直譯器的提示符號。[1] 使用者輸入 $3+4$, Haskell 算出其值 7 並印出. 接著，使用者想得知 1 到 100 的和，Haskell 算出 5050.

[1] 此處的人機互動紀錄擷取自 GHCi (Glasgow Haskell Compiler Interactive). GHC 為目前最被廣泛使用的 Haskell 實作。

演算格式

本書中將使用如下的格式表達（不）等式演算或推論：

$$\begin{aligned} & expr_0 \\ = & \{ reason_0 \} \\ & expr_1 \\ \geqslant & \{ reason_1 \} \\ & expr_2 \\ & \vdots \\ = & expr_n \ . \end{aligned}$$

這是一個 $expr_0 \geqslant expr_n$ 的證明。式子 $expr_0 .. expr_n$ 用具有遞移律的運算子 (如 $(=)$, $(\geqslant)$等等) 串起。放在大括號中的是註解，例如 $reason_0$ 是 $expr_0 = expr_1$ 的原因，$reason_1$ 是 $expr_1 \geqslant expr_2$ 的原因。

根據 van de Snepscheut [1993, p19], 此格式最早由 W.H.J. Feijen 所建議。

上例中的算式只用到 Haskell 已知的函數（如 $(+)$, sum等）。使用者若要自己定義新函數，通常得寫在另一個檔案中，命令 Haskell 直譯器去讀取。例如，我們可把如下的定義寫在一個檔案中：

$$\begin{aligned} & double :: \mathsf{Int} \to \mathsf{Int} \\ & double\ x = x + x \ . \end{aligned}$$

上述定義的第一行是個型別宣告。當我們寫 $e :: t$, 代表 e 具有型別 t. Int 是整數的型別，而箭號 $(\to)$ 為函數型別的建構元。第一行 $double :: \mathsf{Int} \to \mathsf{Int}$ 便是告知電腦我們將定義一個新識別字 $double$, 其型別為「從整數 (Int) 到整數的函數」。[2] 至於該函數的定義本體則在第二行 $double\ x = x + x$, 告知電腦「凡看到 $double\ x$, 均可代換成 $x + x$.」

求值　第 0章曾提及：一套程式語言是設計者看待世界的抽象觀點。程式通常用來表達計算，因此程式語言也得告訴我們：在其假想的世界中，「計算」是怎麼一回事。指令式語言的世界由許多容器般的變數組成，計算是將值在變數之間搬來搬去。邏輯編程中，描述一個問題便是寫下事物間的邏輯關係，計算是邏輯規則「歸結 (resolution)」的附帶效果。共時 (concurrent)程式語言著眼於描述多個同時執行的程式如何透過通道傳遞訊息，計算也靠此達成。 函數語言中，一個程式便是一個數學式，而「計算」便是依照該式子中各個符號的定義，反覆地展開、歸約，直到算出一個「值」為止。這個過程又稱作「求值 (evaluation)」。在 Haskell 直譯器中，若 $double$ 的定義已被讀取，輸入 $double\ (9 \times 3)$, 電腦會算出 54:

$$\begin{aligned} & \mathsf{Main} > double\ (9 \times 3) \\ & 54 \end{aligned}$$

但 54 這個值是怎麼被算出來的？以下是其中一種可能：

[2] Haskell 標準中有多種整數，其中 Int 為有限大小（通常為該電腦中一個字組 (word)）的整數，Integer 則是所謂的大整數或任意精度整數，無大小限制。本書中只使用 Int.

$$
\begin{aligned}
& \mathit{double}\ (9 \times 3) \\
= \quad & \{\ (\times)\ \text{的定義}\ \} \\
& \mathit{double}\ 27 \\
= \quad & \{\ \mathit{double}\ \text{的定義}\ \} \\
& 27 + 27 \\
= \quad & \{\ (+)\ \text{的定義}\ \} \\
& 54\ .
\end{aligned}
$$

上述演算的第一步將 9×3 歸約成 27 — 我們尚未定義 $(\times)$ 與 $(+)$, 但目前只需知道它們與大家所熟悉的整數乘法、加法相同。第二步將 $\mathit{double}\ 27$ 變成 $27 + 27$, 根據的是 double 的定義：$\mathit{double}\ x = x + x$. 最後，$27 + 27$ 理所當然地歸約成 54. 「歸約」一詞由 β-reduction 而來，在此處指將函數本體中的形式參數代換為實際參數。[3] 在上述例子中，我們遇到如 $\mathit{double}\ (9 \times 3)$ 的函數呼叫，先將參數 (9×3) 化簡，再展開函數定義，可說是由內到外的歸約方式。大部分程式語言都依這樣的順序求值，讀者可能也對這種順序較熟悉。

但這並不是唯一的求值順序。我們能否由外到內，先把 double 展開呢？

$$
\begin{aligned}
& \mathit{double}\ (9 \times 3) \\
= \quad & \{\ \mathit{double}\ \text{的定義}\ \} \\
& (9 \times 3) + (9 \times 3) \\
= \quad & \{\ (\times)\ \text{的定義}\ \} \\
& 27 + (9 \times 3) \\
= \quad & \{\ (\times)\ \text{的定義}\ \} \\
& 27 + 27 \\
= \quad & \{\ (+)\ \text{的定義}\ \} \\
& 54\ .
\end{aligned}
$$

以這個順序求值，同樣得到 54.

一般說來，一個式子有許多種可能的求值順序：可能是由內往外、由外往內、或其他更複雜的順序。我們自然想到一個問題：這些不同的順序都會把該式子化簡成同一個值嗎？有沒有可能做出一個式子，使用一個順序會被化簡成 54, 另一個順序化簡成 53?

我們看看如下的例子。函數 three 顧名思義，不論得到什麼參數，都傳回 3；inf 則是一個整數：

$$
\begin{aligned}
& \mathit{three} :: \mathsf{Int} \to \mathsf{Int} \\
& \mathit{three}\ x = 3\ , \\
& \mathit{inf} :: \mathsf{Int} \\
& \mathit{inf} = 1 + \mathit{inf}\ .
\end{aligned}
$$

在指令式語言中，inf 的定義可能會被讀解為「將變數 inf 的值加一」。但函數語言中「變數」的值是不能更改的。此處的意義應是：inf 是一個和 $1 + \mathit{inf}$ 相等的數值。我們來看看 $\mathit{three}\ \mathit{inf}$ 會被為甚麼？如果我們由內往外求值：

[3] Reduction 的另一個常見譯名是「化簡」，然而，許多情況下，一個式子被 reduce 後變得更長而不「簡」，因此本書譯為「歸約」。

$$
\begin{aligned}
& three\ inf \\
= \quad & \{\ inf\ 的定義\ \} \\
& three\ (1+inf) \\
= \quad & \{\ inf\ 的定義\ \} \\
& three\ (1+(1+inf)) \\
= \quad & \{\ inf\ 的定義\ \} \\
& \ldots
\end{aligned}
$$

看來永遠停不下來！但如果我們由外往內，*three inf* 第一步就可變成 3：

$$
\begin{aligned}
& three\ inf \\
= \quad & \{\ three\ 的定義\ \} \\
& 3\ .
\end{aligned}
$$

我們該如何理解、討論這樣的現象呢？

範式與求值順序 為描述、討論相關的現象，我們得非正式地介紹一些術語。用較直觀、不形式化的方式理解，一個式子中「接下來可歸約之處」稱作其歸約點 *(redex)*。例如 $(9\times 3)+(4\times 6)$中，9×3 與 4×6 都是歸約點。如果一個式子已沒有歸約點、無法再歸約了，我們說該式已是個範式 *(normal form)*。

回頭看，經由之前的例子我們已得知：

- 有些式子有範式 (如 $double\ (9\times 3)$ 有個範式 54)，有些沒有（如 *inf*）。
- 同一個式子可用許多順序求值。有些求值順序會碰到範式，有些不會。

給一個式子，我們很自然地希望知道它有沒有值，並算出其值。如果一個式子有很多個範式，我們便難說哪一個才是該式子的「值」。如此一來，立刻衍生出幾個問題。給定一個式子，我們是否能得知它有沒有範式呢？如果有，用哪個求值順序才能得到那個範式？以及，有沒有可能用一個求值順序會得到某範式，換另一個求值順序卻得到另一個範式？

很不幸地，第一個問題的答案是否定的：沒有一套固定的演算法可判定任意一個式子是否有範式。這相當於計算理論中常說到的停機問題 *(halting problem)* — 沒有一個演算法能準確判斷任意一個演算法（對某個輸入）是否能正常終止。

但對於另兩個問題，計算科學中有個重要的 *Church-Rosser* 定理。非常粗略地說，該定理告訴我們：在我們目前討論的這類語言中[4]

- 一個式子最多只有一個範式。
- 如果一個式子有範式，使用由外到內的求值順序可得到該範式。

如此一來，給定任一個式子，我們都可用由外到內的方式算算看。假設一算之下得到（例如）54。用其他的求值順序可能得不到範式，但若有了範式，必定也是 54. 如果由外到內的順序得不到範式，用其他任何順序也得不到。

由於「由外到內」的求值順序有「最能保證得到範式」的性質，又被稱作「範式順序 (*normal order evaluation*)」。「由內到外」的則被稱作「應用順序 (*applicative order evaluation*)」。以本書的目的而言，我們可大致把 Haskell 使用

[4]此處討論的可粗略說是以 λ-calculus 為基礎的函數語言。基於其他概念設計的程式語言當然可能不遵守 Church-Rosser 定理。

的求值方式視為範式順序。但請讀者記得這是個粗略、不盡然正確的說法 — Haskell 實作上使用的求值方式經過了更多最佳化。正式的 λ-calculus 教科書中對於歸約點、求值順序、Church-Rosser 定理等概念會有更精確的定義。

被迫求值　型別 Bool 表示真假，有兩個值 False 和 True。常用的函數 *not* 可定義如下：

$$\begin{aligned} &\mathit{not} :: \mathsf{Bool} \to \mathsf{Bool} \\ &\mathit{not}\ \mathsf{False} = \mathsf{True} \\ &\mathit{not}\ \mathsf{True}\ = \mathsf{False}\ . \end{aligned}$$

此處 *not* 的定義依輸入值不同而寫成兩個條款。這種定義方式在 Haskell 中稱作樣式配對 (*pattern matching*)：False 與 True 在此處都是樣式 (patterns)。遇到這樣的定義時，Haskell 將輸入依照順序與樣式們一個個比對。如果對得上，便傳回相對應的右手邊。本例中，若輸入為 False，傳回值為 True，否則傳回值為 False。

我們來看看 $\mathit{not}\ (5 \leqslant 3)$ 該怎麼求值？若依照範式順序，照理來說應先將 *not* 的定義展開。但若不知 $5 \leqslant 3$ 的值究竟是 False 還是 True, 我們不知該展開哪行定義！因此，要計算 $\mathit{not}\ (5 \leqslant 3)$，也只好先算出 $5 \leqslant 3$ 了：

$$\begin{aligned} &\mathit{not}\ (5 \leqslant 3) \\ =\ &\quad \{\ (\leqslant)\ \text{之定義}\ \} \\ &\mathit{not}\ \mathsf{False} \\ =\ &\quad \{\ \mathit{not}\ \text{之定義}\ \} \\ &\mathsf{True}\ . \end{aligned}$$

求值過程中若必須得知某子算式的值才能決定如何進行，只好先算那個子算式。在 Haskell 中有不少（有清楚定義的）場合得如此，包括遇上 $(\leqslant)$、$(\geqslant)$ 等運算子、樣式配對、**case** 算式（將於第 1.6.1節中介紹）... 等等。

習題 1.1 — 定義一個函數 $\mathit{myeven} :: \mathsf{Int} \to \mathsf{Bool}$，判斷其輸入是否為偶數。你可能用得到以下函數：[a]

$$\begin{aligned} &\mathit{mod} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}\ , \\ &(==)\ :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool}\ . \end{aligned}$$

其中 $\mathit{mod}\ x\ y$ 為 x 除以 y 之餘數，$(==)$ 則用於判斷兩數是否相等。

習題 1.2 — 定義一個函數 $\mathit{area} :: \mathsf{Float} \to \mathsf{Float}$, 給定一個圓的半徑，計算其面積。(可粗略地將 $22/7$ 當作圓周率。)

[a] 此處所給的並非這些函數最一般的型別。

惰性求值　實作上，Haskell 求值的方式還經過了更多的最佳化：例如將歸約過的算式改寫為它的值，避免重複計算。這套求值方式稱為惰性求值 (*lazy evaluation*).

技術上說來，惰性求值和範式順序求值並不一樣。但前者可視為後者的最佳化實作 — 惰性求值的結果必須和範式順序求值相同。因此，在本書之中大部分地方可忽略他們的差異。和惰性求值對偶的是及早求值 (*eager evaluation*)，可視為應用順序求值的實作 — 在呼叫一個函數之前，總是把其參數先算成範式。這也是一般程式語言較常見的計算方法。

本書中談到偏向實作面的議題時會用「惰性求值/及早求值」，在談不牽涉到特定實作的理論時則使用「範式順序求值/應用順序求值」。

1.2 函數定義

考慮如下的函數定義：

$$
\begin{aligned}
&\mathit{smaller} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
&\mathit{smaller}\ x\ y = \textbf{if}\ x \leqslant y\ \textbf{then}\ x\ \textbf{else}\ y\ .
\end{aligned}
$$

我們可粗略地理解為：*smaller* 是一個函數，拿兩個參數 x 與 y，傳回其中較小的那個。如 $\mathit{smaller}\ (\mathit{double}\ 6)\ (3+4)$ 的值為 7.

習題 1.3 — 用前一節介紹的求值順序，將 $\mathit{smaller}\ (\mathit{double}\ 6)\ (3+4)$化簡為範式。

守衛 如果函數本體做的第一件事就是條件判斷，Haskell 提供另一種語法：

$$
\begin{aligned}
&\mathit{smaller} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
&\mathit{smaller}\ x\ y \mid x \leqslant y = x \\
&\qquad\qquad \mid x > y = y\ .
\end{aligned}
$$

這麼寫出的 *smaller* 的行為仍相同：如果 $x \leqslant y$ 成立，傳回 x；如果 $x > y$, 傳回 y。但這種語法較接近一些數學教科書中定義函數的寫法。其中，放在 $x \leqslant y$ 和 $x > y$ 等位置的必須是型別為 Bool 的算式。它們擋在那兒，只在值為 True 的時候才讓執行「通過」，因此被稱為守衛 (*guard*)。如果有三個以上的選項，如下述例子，使用守衛比 **if**..**then**..**else** 更清晰可讀：

$$
\begin{aligned}
&\mathit{sign} :: \mathsf{Int} \to \mathsf{Int} \\
&\mathit{sign}\ x \mid x > 0 = 1 \\
&\qquad\quad \mid x == 0 = 0 \\
&\qquad\quad \mid x < 0 = -1\ .
\end{aligned}
$$

遇到數個守衛時，Haskell 將依照順序嘗試每個選項，直到碰到第一個為真的守衛，然後只執行該項定義。也就是說，即使我們把 *sign* 定義中的 $x == 0$ 改為 $x \geqslant 0$, $\mathit{sign}\ 10$ 的值仍會是 1。若每個守衛都是 False, 程式則將中斷執行（並傳回一個錯誤訊息）。在許多程式中，我們會希望最後一個守衛能捕捉所有的漏網之魚：如果其他條款都不適用，就執行最後一個。一種做法是讓一個守衛是 True。或著，在 Haskell 中有個 *otherwise* 關鍵字可讓定義讀來更口語化些：

$$
\begin{aligned}
&sign :: \mathsf{Int} \to \mathsf{Int} \\
&sign\ x \mid x > 0 \quad\ \ = 1 \\
&\qquad \mid x == 0 \quad\ \ = 0 \\
&\qquad \mid otherwise = -1 \ .
\end{aligned}
$$

事實上，*otherwise* 的定義就是 $otherwise = \mathsf{True}$.

區域識別字 Haskell 提供兩種宣告區域識別字的語法：**let** 與 **where**. 也許最容易理解的方式是用例子說明。

例 1.1. 工讀生每小時的時薪為新台幣 *130* 元。假設一週有五個工作天，每天有八小時上班時間。定義一個函數 $payment :: \mathsf{Int} \to \mathsf{Int}$，輸入某學生工作的週數，計算其薪資。

答. 我們當然可直接用一個式子算出薪資。但為清楚起見，我們可用兩個區域識別字 *days* 和 *hours*, 分別計算該學生工作的日數和時數。如果用 **let**, 可這麼做。

$$
\begin{aligned}
&payment :: \mathsf{Int} \to \mathsf{Int} \\
&payment\ weeks = \mathbf{let}\ days\ \ = 5 \times weeks \\
&\qquad\qquad\qquad\quad\ \ hours = 8 \times days \\
&\qquad\qquad\quad\ \ \mathbf{in}\ 130 \times hours \ .
\end{aligned}
$$

let 算式的語法為

$$
\begin{aligned}
&\mathbf{let}\ x_1 = e_1 \\
&\quad\ \ x_2 = e_2 \ldots . \\
&\mathbf{in}\ e \ .
\end{aligned}
$$

其中 e 為整個算式的值，而 x_1, x_2 等等為區域識別字。兩個區域識別字的有效範圍包括 e, 以及 e_1 與 e_2。

另一種語法是 **where** 子句。若用它定義 *payment*, 可寫成這樣：

$$
\begin{aligned}
&payment :: \mathsf{Int} \to \mathsf{Int} \\
&payment\ weeks = 130 \times hours \ , \\
&\quad \mathbf{where}\ hours = 8 \times days \\
&\qquad\quad\ \ days\ \ = 5 \times weeks \ .
\end{aligned}
$$

□

該用 **let** 或是 **where**? 大部份時候這可依個人習慣，看寫程式的人覺得怎麼說一件事情比較順。使用 **let** 時，敘事的順序是由小到大，先給「工作日數」、「工作時數」等小元件的定義，再用他們組出最後的式子 $130 \times hours$。使用 **where** 時則是由大到小，先說我們要算「工作時數乘以 130」，然後補充「其中，工作時數的定義是...」。

但，Haskell 之所以保留了兩種語法，是為了因應不同的用途。語法上，**let** 是一個算式，可出現在算式中。如下的算式是合法的，其值為 24:

$$(1+1)\times(\textbf{let } y = \mathit{double}\ 5 \textbf{ in } y+2)\ .$$

where 子句的一般語法則如下例所示：

$$\begin{aligned} &f\ x_0 = d_0 \\ &\quad \textbf{where } y_0 = e_0 \\ &f\ x_1 = d_1 \\ &\quad \textbf{where } y_1 = e_1\ . \end{aligned}$$

由語法設計上即可看出，子句 **where** $y_0 = e_0$ 只能放在 $f\ x_0 = \ldots$ 的一旁當作補述，不能出現在 d_0 之中。這個例子中，y_0 的有效範圍含括 d_0 與 e_0。另，e_0 可以使用 x_0.

算式中只能用 **let**. 相對地，也有些只能使用 **where** 的場合。我們來看我們來看一個只能使用 **where** 的例子：

例 1.2. 延續例 *1.1*。今年起，新勞動法規規定工作超過 *19* 週的工讀生必須視為正式雇員，學校除了薪資外，也必須付給勞保、健保費用。學校需負擔的勞健保金額為雇員薪資的百分之六。請更新函數 *payment*, 輸入某工讀生工作週數，計算在新規定之下，學校需為工讀生付出的總額。

答. 一種可能寫法是先使用守衛，判斷工作週數是否大於 19：

$$\begin{aligned} &\mathit{payment} :: \mathsf{Int} \to \mathsf{Int} \\ &\mathit{payment}\ \mathit{weeks} \mid \mathit{weeks} > 19 = \mathit{round}\ (\mathit{fromIntegral}\ \mathit{baseSalary} \times 1.06) \\ &\qquad\qquad\quad \mid \mathit{otherwise}\ \ = \mathit{baseSalary}\ , \\ &\textbf{where } \mathit{baseSalary} = 130 \times \mathit{hours} \\ &\qquad\quad \mathit{hours}\ \ = 8 \times \mathit{days} \\ &\qquad\quad \mathit{days}\ \ = 5 \times \mathit{weeks}\ . \end{aligned}$$

在 **where** 子句中，我們先算出不含勞健保費用的薪資，用識別字 *baseSalary* 表達。如果 *weeks* 大於 19, 我們得將 *baseSalary* 乘以 1.06；否則即傳回 *baseSalary*. 函數 *fromIntegral* 把整數轉為浮點數，*round* 則把浮點數四捨五入為整數。請注意：兩個被守衛的算式都用到了 *baseSalary* — **where** 子句中定義的識別字是可以跨越守衛的。相較之下，**let** 算式只能出現在等號的右邊，而在守衛 $\mathit{weeks} > 19 = \ldots$ 之後出現的 **let** 所定義出的區域識別字，顯然無法在 $\mathit{otherwise} = \ldots$ 之中被使用，反之亦然。 □

巢狀定義 **let** 算式之中還可有 **let** 算式，**where** 子句中定義的識別字也可有自己的 **where** 子句。我們看看兩個關於 **let** 的例子：

例 1.3. 猜猜看 *nested* 和 *recursive* 的值分別是什麼。將他們載入 *Haskell* 直譯器，看看和你的猜測是否相同。

$$\begin{aligned} &\mathit{nested} :: \mathsf{Int} \\ &\mathit{nested} = \textbf{let } x = 3 \\ &\qquad\quad \textbf{in } (\textbf{let } x = 5 \\ &\qquad\qquad\quad \textbf{in } x+x)+x\ , \end{aligned} \qquad \begin{aligned} &\mathit{recursive} :: \mathsf{Int} \\ &\mathit{recursive} = \textbf{let } x = 3 \\ &\qquad\qquad\quad \textbf{in let } x = x+1 \\ &\qquad\qquad\qquad \textbf{in } x\ . \end{aligned}$$

一級公民

在程式語言中，若說某物/某概念是一級公民 (*first-class citizen*)或「一級的」，通常指它和其他事物被同等對待：如果其他事物可被當作參數、可被當作傳回值、可被覆寫... 那麼它也可以。這是一個沒有嚴格形式定義的說法，由 Christopher Strachey 在 1960 年代提出，可用在型別、值、物件、模組... 等等之上。

例如：OCaml 是個有「一級模組」的語言，因為 OCaml 模組也可當作參數，可定義從模組到模組的函數（OCaml 中稱之為 functor）。在 C 語言之中函數是次級的，因為函數不能當參數傳（能傳的是函數的指標，而非函數本身）。Strachey 指出，在 Algol 中實數是一級的，而程序是次級的。

答. *nested* 的值是 13，因為 $x+x$ 之中的 x 在 **let** $x=5$ 的範圍中, 而 $..+x$ 中的 x 則在 **let** $x=3$ 的範圍中。[5] 至於 *recursive* 的值，關鍵在於 $x=x+1$ 中右手邊的 x 指的是哪個。若是 $x=3$ 的那個 x, 整個算式的值將是 4. 若 $x=x+1$ 中，等號右手邊的 x 也是左手邊的 x, *recursive* 就是 $((..)+1)+1$, 沒有範式。這兩種設計都有其道理。Haskell 選了後者：在 **let** $x=e$ **in**... 之中，x 的有效範圍包括 e. 因此 *recursive* 在 Haskell 中會無窮地求值下去。但也有些函數語言中的 **let** 採用前者的設計。通常這類語言中會另有一個 **letrec** 結構，和 Haskell 的 **let** 功能相同。 □

1.3 高階函數

目前為止，我們看過由整數到整數的函數、由整數到真假值的函數... 那麼，可以有由函數到函數的函數嗎？函數語言將函數視為重要的構成元件，因此函數也被視為一級公民。如果整數、真假值... 可以當作參數、可以被函數傳回，函數當然也可以。一個「輸入或輸出也是函數」的函數被稱為高階函數 (*higher order function*)。Haskell 甚至設計了許多鼓勵我們使用高階函數的機制。本書中我們將見到許多高階函數。其實，我們已經看過一個例子了。

Currying 回顧 *smaller*的定義：

$$\begin{aligned} &\mathit{smaller} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ &\mathit{smaller}\ x\ y = \textbf{if}\ x \leqslant y\ \textbf{then}\ x\ \textbf{else}\ y \ . \end{aligned}$$

1.2 節中說「*smaller* 是一個函數，拿兩個參數 x 與 y」。但這僅是口語上方便的說法。事實上，在 Haskell 中（如同在 λ-calculus 中），所有函數都只有一個參數。函數 *smaller* 其實是一個傳回函數的函數：

- *smaller* 的型別 $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$ 其實應看成 $\mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$：這個函數拿到一個 Int 後，會傳回一個型別為 $\mathsf{Int} \to \mathsf{Int}$ 的函數。
- *smaller* 3 的型別是 $\mathsf{Int} \to \mathsf{Int}$。這個函數還可拿一個 Int 參數，將之和 3 比大小，傳回較小的那個。

[5] 在各種語言中，範圍的設計都是為了給程式員方便：在寫子算式時，可不用擔心是否與外層的識別字撞名。在教學時，我們難免舉各種撞名的例子作為說明。若把這些刁鑽例子當作考題，就是違反設計者本意的發展了。

- *smaller* 3 4 是一個 Int。它其實是將函數 *smaller* 3 作用在 4 之上。也就是說，*smaller* 3 4 其實應看成 (*smaller* 3) 4. 根據定義，它可展開為 **if** $3 \leqslant 4$ **then** 3 **else** 4，然後化簡為 3.

習題 1.4 — 將 *smaller* 的定義鍵入一個檔案，載入 Haskell 直譯器中。

1. *smaller* 3 4 的型別是什麼？在 GHCi 中可用 `:t e` 指令得到算式 *e* 的型別。
2. *smaller* 3 的型別是什麼？
3. 在檔案中定義 $st3 = smaller\ 3$. 函數 *st3* 的型別是什麼？
4. 給 *st3* 一些參數，觀察其行為。

「用『傳回函數的函數』模擬『接收多個參數的函數』」這種做法稱作 *currying*.[6] Haskell 鼓勵大家使用 currying — 它的內建函數大多依此配合設計，語法設計上也很給 currying 方便。當型別中出現連續的 $(\to)$ 時，預設為往右邊結合，例如 $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$ 應看成 $\mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$. 這使得「傳回函數的函數」容易寫。而在值的層次，連續的函數應用往左結合。例如，(*smaller* 3) 4 可寫成 *smaller* 3 4。這讓我們能很方便地將參數一個個餵給 curried 函數。

另一方面，如果我們想使用 *double* 兩次，計算 *x* 的四倍，應寫成 *double* (*double x*). 若寫 *double double x* ，會被視為 (*double double*) *x* — *double* 作用在自身之上，而這顯然是個型別錯誤。

我們再看一個使用 currying 的練習：

例 1.4. 給定 a, b, c, x, 下述函數 *poly* 計算 $ax^2 + bx + c$:

$$\begin{aligned} &poly :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ &poly\ a\ b\ c\ x = a \times x \times x + b \times x + c \ . \end{aligned}$$

請定義一個函數 $poly_1$, 使得 $poly_1\ x = x^2 + 2x + 1$. 函數 $poly_1$ 需使用 *poly*.

答. 一種作法是：

$$\begin{aligned} &poly_1 :: \mathsf{Int} \to \mathsf{Int} \\ &poly_1\ x = poly\ 1\ 2\ 1\ x \ . \end{aligned}$$

但這相當於 $poly_1\ x = (poly\ 1\ 2\ 1)\ x$ — *poly* 拿到 *x* 後，立刻把 *x* 傳給 *poly* 1 2 1 這個函數。因此 $poly_1$ 可更精簡地寫成：

$$\begin{aligned} &poly_1 :: \mathsf{Int} \to \mathsf{Int} \\ &poly_1 = poly\ 1\ 2\ 1 \ . \end{aligned}$$

兩種寫法都有人使用。有提及 *x* 的寫法著重於描述拿到參數 *x* 之後要對它進行什麼操作。而省略 *x* 的寫法則是在函數的層次上思考：我們要定義一個函數，稱作 $poly_1$。這個函數是什麼呢？就是 *poly* 拿到 1, 2, 1 之後傳回的那個函數。

[6] Currying 為動名詞，形容詞則為 *curried*。此詞來自於邏輯學家 Haskell B. Curry 的姓氏。詳見第 1.11 節。

如果我們想用 $poly$ 定義出另一個函數 $poly_2\ a\ b\ c = a \times 2^2 + b \times 2 + 2$ 呢？最好理解的可能是不怎麼意外的寫法：

$$poly :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$$
$$poly_2\ a\ b\ c = poly\ a\ b\ c\ 2\ .$$

我們可以用一些技巧使 a, b, 和 c不出現在定義中，但如此得到的程式並不會更好懂。 □

二元運算子 在進入其他主題前，我們交代一些語法細節。Haskell 鼓勵 currying 的使用，也把二元運算子都設計成 curried 的。例如加法的型別是 $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. Haskell 也配套設計了種種關於二元運算子的特殊語法，希望讓它們更好用。但這些語法規則的存在都僅是為了方便我們寫出（主觀上）語法漂亮的程式，而不是非有不可、非學不可的規定。假設某二元運算子 $(\oplus)$ 的型別是 $a \to b \to c$, $(x\oplus)$ 是給定了 $(\oplus)$ 的第一個參數後得到的函數；$(\oplus y)$ 則是給定了 $(\oplus)$ 的第二個參數後得到的函數：[7]

$$(x\oplus)\ y = x \oplus y \qquad \{\,(x\oplus) \text{ 的型別為 } b \to c;\,\}$$
$$(\oplus y)\ x = x \oplus y \qquad \{\,(\oplus y) \text{ 的型別為 } a \to c.\,\}$$

例如：

- $(2\times)$ 和 $(\times 2)$ 都是把一個數字乘以二的函數；
- $(/2)$ 則把輸入數字除以二；
- $(1/)$ 計算輸入數字的倒數。

名字以英文字母開頭的函數預設為前序的。例如，計算餘數的函數 mod 使用時得寫成 $mod\ 6\ 4$。若把它放在「倒引號 (backquote)」中，表示將其轉為中序 — 如果我們比較喜歡把 mod 放到中間，可以寫成 $6\ `mod`\ 4$. 首字元非英文字母的函數（如 $(+)$, $(/)$ 等）則會被預設為中序的二元運算子。若把一個中序二元運算子放在括號中，表示將其轉為前序運算子。例如，$(+)\ 1\ 2$ 和 $1+2$ 的意思相同。

在 Haskell 的設計中，函數應用的優先順序比中序運算子高。因此 $double\ 3+4$ 會被視作 $(double\ 3)+4$, 而不是 $double\ (3+4)$. 將中序運算子放在括號中也有「讓它不再是個中序運算子，只是個一般識別字」的意思。例如算式 $f+x$ 中，f 和 x 是中序運算子 $(+)$ 的參數。但在 $f\ (+)\ x$ 中，$(+)$ 和 x 都是 f 的參數（這個式子可以讀解為 $(f\ (+))\ x$）。

以函數為參數 下述函數 $square$ 計算輸入的平方：

$$square :: \mathsf{Int} \to \mathsf{Int}$$
$$square\ x = x \times x\ .$$

我們可另定義一個函數 $quad :: \mathsf{Int} \to \mathsf{Int}$，把 $square$ 用兩次，使得 $quad\ x$ 算出 x^4.

[7] 根據 Hudak et al. [2007]，此種「切片」(*sectioning*) 語法最早見於 David Wile 的博士論文。後來被包括 Richard Bird 在內的 IFIP WG 2.1 成員使用，並由 David A. Turner 實作在他的語言 Miranda 中。

$$quad :: \mathsf{Int} \to \mathsf{Int}$$
$$quad\ x = square\ (square\ x)\ .$$

但，「把某函數用兩次」是個常見的編程模式。我們能不能把 *quad* 與 *square* 抽象掉，單獨談「用兩次」這件事呢？下面的函數 *twice* 把參數 f 在 x 之上用兩次：

$$twice\ \ :: (a \to a) \to (a \to a)$$
$$twice\ f\ x = f\ (f\ x)\ .$$

有了 *twice*, 我們可以這麼定義 *quad*:

$$quad :: \mathsf{Int} \to \mathsf{Int}$$
$$quad = twice\ square\ .$$

函數 *twice* 是本書中第一個「以函數為參數」的函數。我們可看到「讓函數可作為參數」對於抽象化是有益的：我們可以把「做兩次」這件事單獨拿出來說，把「做什麼」抽象掉。

「函數可以當作參數」意味著我們可以定義作用在函數上的運算子。*twice* 就是這麼一個運算子：它拿一個函數 f，把它加工一下，做出另一個函數（後者的定義是把 f 用兩次）。

參數式多型 函數 *twice* 也是本書中第一個多型函數。在 Haskell 的型別中，小寫開頭的識別字(如其中的 a)是型別層次的參數。讀者可想像成在 *twice* 的型別最外層有一個省略掉的 $\forall a$. 也就是說, *twice* 的完整型別是 $\forall a \cdot (a \to a) \to (a \to a)$ — 對所有的型別 a, *twice* 都可拿一個型別為 $a \to a$ 的函數，然後傳回一個型別為 $a \to a$ 的函數。

在 *twice* 的型別 $(a \to a) \to (a \to a)$ 中，

- 第一個 $a \to a$ 是參數 f 的型別，
- 在第二個 $a \to a$ 中，第一個 a 是參數 x 的型別，
- 第二個 a 則是整個計算結果的型別。

參數 f 的型別必須是 $a \to a$: 輸出入型別必須一樣，因為 f 的結果必須可當作 f 自己的輸入。

在 *twice* 被使用時，型別參數 a 會依照上下文被特化 (*instantiate*) 成別的型別。例如 *twice square* 中，因為 *square* 的型別是 $\mathsf{Int} \to \mathsf{Int}$, 這一個 *twice* 的型別變成了 $(\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$ — a 被特化成 Int. 若某函數 k 的型別是 $\mathsf{Float} \to \mathsf{Float}$, 在 *twice k* 中，*twice* 的型別是 $(\mathsf{Float} \to \mathsf{Float}) \to (\mathsf{Float} \to \mathsf{Float})$。同一個函數 *twice* 可能依其上下文而有許多不同的型別，但都是 $(a \to a) \to (a \to a)$ 的特例。「一段程式可能有許多不同型別」的現象稱作多型 (*polymorphism*)。多型又有許多種類，此處為其中一種。詳情見... [todo:]

習題 1.5 — 為何 *twice* 的型別不可以是 $(a \to b) \to (a \to b)$?

例 1.5. $forktimes\ f\ g\ x = f\ x \times g\ x\ .$

算式 $forktimes\ f\ g\ x$ 把 $f\ x$ 和 $g\ x$ 的結果乘起來。

1. 請想想 $forktimes$ 的型別該是什麼?
2. 試定義函數 $compute :: \mathsf{Int} \to \mathsf{Int}$, 使用 $forktimes$ 計算 $x^2+3\times x+2$。**提示**: $x^2+3\times x+2=(x+1)\times(x+2)$.

答. 如同 $twice, forktimes$ 可以有很多型別，但都應該是 $(a\to\mathsf{Int})\to(a\to\mathsf{Int})\to a\to\mathsf{Int}$ 的特例：在 $forktimes\ f\ g\ x$ 中，f 和 g 的型別可以是 $a\to\mathsf{Int}$, 其中 a 可以是任何型別 a，而 x 的型別必須也是同一個 a. 函數 $compute$ 可定義如下：

$$\begin{aligned} &compute :: \mathsf{Int}\to\mathsf{Int} \\ &compute = forktimes\ (+1)\ (+2)\ . \end{aligned}$$

其中 $forktimes$ 型別中的 a 被特化為 Int. □

如前所述，$forktimes\ f\ g\ x$ 把 $f\ x$ 和 $g\ x$ 的結果乘起來。但，一定得是乘法嗎？我們當然可以再多做一點點抽象化。

例 1.6. 考慮函數 $lift_2\ h\ f\ g\ x = h\ (f\ x)\ (g\ x)$.

1. $lift_2$ 的型別是什麼?
2. 用 $lift_2$ 定義 $forktimes$.
3. 用 $lift_2$ 計算 $x^2+3\times x+2$.

答. 我們把 $lift_2$ 最泛用的型別和其定義重複如下：

$$\begin{aligned} &lift_2 :: (a\to b\to c)\to(d\to a)\to(d\to b)\to d\to c\ . \\ &lift_2\ h\ f\ g\ x = h\ (f\ x)\ (g\ x)\ . \end{aligned}$$

有了 $lift_2, forktimes$ 可定義為：

$$\begin{aligned} &forktimes :: (a\to\mathsf{Int})\to(a\to\mathsf{Int})\to a\to\mathsf{Int} \\ &forktimes = lift_2\ (\times)\ , \end{aligned}$$

請讀者觀察：$lift$ 型別中的 a, b, c 都特化成 Int, d 則改名為 a.

我們也可用 $lift_2$ 定義 $compute$:

$$\begin{aligned} &compute :: \mathsf{Int}\to\mathsf{Int} \\ &compute = lift_2\ (\times)\ (+1)\ (+2)\ . \end{aligned}$$

函數 $lift_2$ 可以看作一個作用在二元運算子上的運算子，功用是把二元運算子「提升」到函數層次。例如，原本 $(\times)$ 只能拿兩個 Int 當作參數，(例：1×2 是「把 1 和 2 乘起來」)，但現在 $lift_2\ (\times)$ 可將函數 $(+1)$ 和 $(+2)$ 當參數了，意思為「把 $(+1)$ 和 $(+2)$ 的結果乘起來」。 □

1.4 函數合成

拿到一個函數 f，我們能做的基本操作包括把 f 作用在某個參數上、把 f 傳給別的函數... 此外，另一個常用的基本操作是將 f 和別的函數合成 (*compose*) 為

一個新函數。[8] 「合成」運算子在 Haskell 中寫成 $(\cdot)$. 這個運算子的形式定義如下（我們先看定義本體，待會兒再看型別）：

$$(f \cdot g)\, x = f\ (g\, x)\ .$$

若用口語說，$f \cdot g$ 是將 f 和 g兩個函數「串起來」得到的新函數：輸入 x 先丟給 g, 後者算出的結果再傳給 f.

例 1.7. *square* · *double* 與 *double* · *square* 都是由 Int 到 Int 的函數. 直覺上，前者把輸入先給 *double*, 其結果再給 *square*。後者則反過來。如何了解它們的行為？既然它們是函數，我們便餵給它們一個參數，看看會展開成什麼！兩者分別展開如下：

$$\begin{aligned} & (square \cdot double)\, x \\ = & \quad \{(\cdot)\ 的定義\} \\ & square\ (double\ x) \\ = & (x+x) \times (x+x)\ , \end{aligned} \qquad \begin{aligned} & (double \cdot square)\, x \\ = & \quad \{(\cdot)\ 的定義\} \\ & double\ (square\ x) \\ = & (x \times x) + (x \times x)\ . \end{aligned}$$

所以，如果輸入為 x, $(square \cdot double)\, x$ 計算 $(2x)^2$; $(double \cdot square)\, x$ 則是 $2x^2$.

但，並不是所有函數都可以串在一起：$f \cdot g$ 之中，g 的輸出型別和 f 的輸入型別必須一致才行。運算子 $(\cdot)$ 包括型別的完整定義為：

$$\begin{aligned} & (\cdot) :: (b \to c) \to (a \to b) \to a \to c \\ & (f \cdot g)\, x = f\ (g\, x)\ . \end{aligned}$$

如果 g 的型別是 $a \to b$, f 的型別是 $b \to c$, 將他們串接起來後，便得到一個型別為 $a \to c$ 的函數。

有了 $(\cdot)$, 函數 *twice* 可以定義如下：

$$\begin{aligned} & twice\ \ :: (a \to a) \to (a \to a) \\ & twice\, f = f \cdot f\ \ . \end{aligned}$$

確實，根據 $(\cdot)$ 的定義，$twice\, f\, x = (f \cdot f)\, x = f\ (f\, x)$，和 *twice* 原來的定義相同。

為了討論函數合成的性質，我們先介紹一個函數 *id*:

$$\begin{aligned} & id :: a \to a \\ & id\, x = x\ . \end{aligned}$$

它又稱作單位函數或恆等函數。這是一個似乎沒有在做什麼的函數：給任何輸入，*id* 都原封不動地把它送到輸出 — 這也反映在他的型別 $a \to a$ 上。這個函數有什麼重要性呢？原來，$(\cdot)$ 滿足結合律，並且以 *id* 為單位元素（這也是「單位函數」這名字的由來）：

$$\begin{aligned} id \cdot f\ &= f\ = f \cdot id\ , \\ (f \cdot g) \cdot h\ &= f \cdot (g \cdot h)\ . \end{aligned}$$

[8] Robert Glück 認為函數上應有三個基本操作：函數應用、函數合成、以及求一個函數的反函數。前兩者已經提到，第三者則是大部分語言欠缺的。

用數學術語來說的話，id 與 $(\cdot)$ 形成一個幺半群 (*monoid*)。函數 id 的重要性就如同 0 在代數中的重要性（0 與 $(+)$ 也是一個幺半群）。我們在許多計算、證明中都會見到它。

以下我們試著證明 $(\cdot)$ 的結合律。我們想論證 $(f \cdot g) \cdot h = f \cdot (g \cdot h)$，但該如何下手？該等式的等號左右兩邊都是函數。當我們說兩個整數相等，意思很清楚：如果等號左邊是 0, 右邊也是 0; 如果左邊是 1, 右邊也是 1... 但說兩個函數「相等」，是什麼意思呢？

定義 1.8 (外延相等 (extensional equality))**.** 給定兩個型別相同的函數 f 和 g, 當我們說它們外延相等 (extensionally equal), 意思是給任何一個輸入, f 和 g 都算出相等的輸出。也就是：$(\forall x \cdot f\ x = g\ x)$.

本書中，當我們寫兩個函數相等 $(f = g)$ 時，指的便是外延相等，除非例外註明。

在外延相等的假設下，證明 $(f \cdot g) \cdot h = f \cdot (g \cdot h)$ 也就是證明對任何一個 x, $((f \cdot g) \cdot h)\ x = (f \cdot (g \cdot h))\ x$ 均成立. 我們推論如下：

$$
\begin{aligned}
& ((f \cdot g) \cdot h)\ x \\
= \quad & \{\ (\cdot) \text{ 的定義}\ \} \\
& (f \cdot g)\ (h\ x) \\
= \quad & \{\ (\cdot) \text{ 的定義}\ \} \\
& f\ (g\ (h\ x)) \\
= \quad & \{\ (\cdot) \text{ 的定義}\ \} \\
& f\ ((g \cdot h)\ x)\ . \\
= \quad & \{\ (\cdot) \text{ 的定義}\ \} \\
& (f \cdot (g \cdot h))\ x\ .
\end{aligned}
$$

既然 $(f \cdot g) \cdot h = f \cdot (g \cdot h)$，我們便可統一寫成 $f \cdot g \cdot h$, 不用加括號了。

習題 1.6 — 證明 $id \cdot f = f = f \cdot id$.

合成 $(\cdot)$ 也是一個中序運算子。和其他中序運算子一樣，其優先性低於函數應用。因此，當我們寫 $f \cdot g\ x$, 指的是 $f \cdot (g\ x)$ — $g\ x$ 為一個函數，和 f 合成，而不是 $(f \cdot g)\ x$ （後者根據 $(\cdot)$ 的定義，是 $f\ (g\ x)$）。

例 1.9. 下列程式中，有些是合法的 *Haskell* 式子、有些則有型別錯誤。對每個程式，如果它是合法的，請找出它的型別，並說說看該程式做什麼。如果有型別錯誤，請簡述為什麼。

1. $square \cdot smaller\ 3$;
2. $smaller\ 3 \cdot square$;
3. $smaller\ (square\ 3)$;
4. $smaller \cdot square\ 3$.

答. 前三者都是 $\mathsf{Int} \to \mathsf{Int}$。

1. 根據 $(\cdot)$ 的定義，$(square \cdot smaller\ 3)\ x = square\ (smaller\ 3\ x)$. 因此 $square \cdot smaller\ 3$ 是一個函數，將其輸入和 3 比較，取較小者的平方。

2. $(smaller\ 3 \cdot square)\ x = smaller\ 3\ (square\ x)$. 因此它讀入 x, 並在 3 或 $x \uparrow 2$ 之中選較小的那個。
3. $smaller\ (square\ 3)$ 是一個函數，讀入 x 之後，選擇 x 與 $3 \uparrow 2$ 之中較小的那個。
4. $smaller \cdot square\ 3$ 有型別錯誤：$square\ 3$ 不是一個函數（而是一個整數），無法和 $smaller$ 合成。

□

函數應用運算子 本書中有些時候會將許多函數組合成一串，例如 $square \cdot double \cdot (+1) \cdot smaller\ 3$ 。由於函數應用的優先順序比一般二元運算元高，把上述式子應用在參數 5 之上時得寫成

$$(square \cdot double \cdot (+1) \cdot smaller\ 7)\ 5\ ,$$

(這個式子的值為 $(2 \times (5+1))^2$ 。) 每次都得加一對括號似乎有些累贅。Haskell 另有一個運算子 $(\$)$, 唸作 "apply", 代表函數應用：

$$\begin{aligned}&(\$) :: (a \to b) \to a \to b\\ &f\ \$\ x = f\ x\ .\end{aligned}$$

$f\ \$\ x$ 和 $f\ x$ 意思一樣。那麼我們為何需要這個運算子呢？原因之一是 $(\$)$ 的優先度比 $(\cdot)$ 低，因此上式可省去括號改寫如下：

$$square \cdot double \cdot (+1) \cdot smaller\ 7\ \$\ 5\ .$$

運算子 $(\$)$ 的另一個重要意義是：「函數應用」這個動作有了符號，成為可以獨立討論的事物。例如，$(\$)$ 可以當作參數。一個這麼做的例子是習題 1.11。

常量函數 既然介紹了 id, 本節也順便介紹一個以後將使用到的基本組件。給定 x 之後，函數 $const\ x$ 是一個 不論拿到什麼函數，都傳回 x的函數。函數 $const$ 的定義如下：

$$\begin{aligned}&const :: a \to b \to a\\ &const\ x\ y = x\ .\end{aligned}$$

第 1.1節開頭的範例 $three$ 可定義為 $three = const\ 3$.

「無論如何都傳回 x」聽來好像是個沒用的函數，但和 id 一樣，我們日後會看到它在演算、證明中時常用上。事實上，組件邏輯理論告訴我們：所有函數都可以由 id, $const$, 和下述的 $subst$ 三個函數組合出來。

$$\begin{aligned}&subst :: (a \to b \to c) \to (a \to b) \to (a \to c)\\ &subst\ f\ g\ x = f\ x\ (g\ x)\ .\end{aligned}$$

1.5 λ 算式

雖說函數是一級市民，在本書之中，目前為止，仍有一項功能是其他型別擁有、函數卻還沒有的：寫出一個未命名的值的能力。整數、真假值都能不經命名、直接寫在算式中，例如，我們可寫 $smaller\ (square\ 3)\ 4$, 而不需要先定義好

$$\begin{aligned} &num_1, num_2 :: \mathsf{Int} \\ &num_1 = 3 \\ &num_2 = 4\ \ , \end{aligned}$$

才能說 $smaller\ (square\ num_1)\ num_2$. 但使用函數時，似乎非得先給個名字，才能使用它：

$$\begin{aligned} &square :: \mathsf{Int} \to \mathsf{Int} \\ &square\ x = \ldots\ \ , \\ &quad = twice\ square\ \ . \end{aligned}$$

如果為了某些原因（例如，如果在我們的程式中 *square* 只會被用到一次），我們不想給 *square* 一個名字，我們能不能直接把它寫出來呢？

λ 算式便是允許我們這麼做的語法。以直觀的方式解釋，$\lambda x \to e$ 便是一個函數，其中 x 是參數，e 是函數本體。例如，$(\lambda x \to x \times x)$ 是一個函數，計算其輸入 (x) 的平方。如果我們不想給 *square* 一個名字, 我們可將 *quad* 定義為：

$$quad = twice\ (\lambda x \to x \times x)\ \ .$$

寫成 λ 算式的函數也可直接作用在參數上，例如 $(\lambda x \to e_1)\ e_2$。這個式子歸約的結果通常表示為 $e_1\ [e_2 / x]$, 意思是「將 e_1 之中的 x 代換為 e_2」。例如，算式 $(\lambda x \to x \times x)\ (3+4)$ 可歸約為 $(3+4) \times (3+4)$.

例 1.10. 以下是一些 λ 算式的例子：

- 函數 $(\lambda x \to 1 + x)$ 把輸入遞增一和 $(1+)$ 相同。其實，把 $(1+)$ 的語法糖去掉後，得到的就是這個 λ 算式。
- $(\lambda x \to \lambda y \to x + 2 \times x \times y)$ 是一個傳回 λ 算式的函數。
  - $(\lambda x \to \lambda y \to x + 2 \times x \times y)\ (3+4)$ 可歸約為 $(\lambda y \to (3+4) + 2 \times (3+4) \times y)$。注意 λx 不見了，函數本體中的 x 被代換成 $3+4$，$\lambda y \to ..$ 則仍留著。
  - $(\lambda x \to \lambda y \to x + 2 \times x \times y)\ (3+4)\ 5$ 可歸約為 $(3+4) + 2 \times (3+4) \times 5$.
- 由於傳回函數的函數是常見的，*Haskell* (如同 λ*-calculus*) 提供較短的語法。上述例子中的函數也可簡寫成：$(\lambda x\, y \to x + 2 \times x \times y)$.
- 函數也可以當參數。例如，$(\lambda x \to x\ 3\ 3)\ (+)$ 可歸約為 $(+)\ 3\ 3$, 或 $3+3$.
- 以下是 $(\lambda f\ x \to f\ x\ x)\ (\lambda y \to 2 \times y)\ 3$ 的求值過程：

$$\begin{aligned} &(\lambda f\ x \to f\ x\ x)\ (\lambda y\ z \to 2 \times y + z)\ 3 \\ &= (\lambda x \to (\lambda y \to 2 \times y + z)\ x\ x)\ 3 \\ &= (\lambda y \to 2 \times y + z)\ 3\ 3 \\ &= 2 \times 3 + 3 \\ &= 9\ \ . \end{aligned}$$

- 在 $\lambda x \to e$ 之中，x 是範圍限於 e 的區域識別字。因此：

$$\begin{aligned}
&(\lambda f\, x \to x + f\, x)\ (\lambda x \to x + x)\ 3 \\
&= (\lambda x \to x + (\lambda x \to x + x)\ x)\ 3 \\
&= 3 + (\lambda x \to x + x)\ 3 \\
&= 3 + 3 + 3 \\
&= 9\ .
\end{aligned}$$

有了 λ 算式後，函數 *smaller* 又有另一種寫法：

$$\begin{aligned}
&\mathit{smaller} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
&\mathit{smaller} = \lambda x\, y \to \textbf{if}\ x \leqslant y\ \textbf{then}\ x\ \textbf{else}\ y\ .
\end{aligned}$$

事實上，λ 算式可視為更基礎的機制 — 目前為止我們所介紹的種種語法結構都僅是 λ 算式的語法糖，都可展開、轉譯為 λ 算式。Haskell 的 λ 算式源於一套稱為 λ 演算 (λ *calculus*) 的形式語言 — 這是一個為了研究計算本質而發展出的理論，也是函數語言的理論核心。我們將在爾後的章節中做更詳盡的介紹。[todo: which?]

1.6 簡單資料型態

藉由一些例子，我們已經看過 Haskell 的一些數值型別：Int, Float 等等。在本節中我們將簡短介紹我們將用到的一些其他型別。

1.6.1 布林值

布林值 (Boolean)常用於程式中表達真和假。在 Haskell 中，我們可假想有這樣的一個型別定義：

$$\textbf{data}\ \mathsf{Bool} = \mathsf{False} \mid \mathsf{True}\ .$$

其中，**data** 是 Haskell 宣告新資料型別的保留字。上述定義可用口語描述成「定義一個稱作 Bool 的新資料型別，有兩個可能的值，分別為 False 和 True.」False 和 True 是型別 Bool 的唯二兩個建構元 — 任何型別為 Bool 的值，如果有正規式，必定是它們兩者之一。在 Haskell 之中，建構元必須以大寫英文字母或冒號 (:) 開頭。

樣式配對 有了資料，我們來看看怎麼定義該型別上的函數。以布林值為輸入的函數中，最簡單又常用的可能是 *not*:

$$\begin{aligned}
&\mathit{not} :: \mathsf{Bool} \to \mathsf{Bool} \\
&\mathit{not}\ \mathsf{False} = \mathsf{True} \\
&\mathit{not}\ \mathsf{True}\ = \mathsf{False}\ .
\end{aligned}$$

這和我們的的直覺理解一致：*not* False 是 True, *not* True 是 False. 我們看到這個定義寫成兩行（正式說來是兩個「子句」），每一個子句分別對應到 Bool 的

一個可能的值。以下則是邏輯上的「且」和「或」（分別寫作 $(\wedge)$與 $(\vee)$）的定義：[9]

$$
\begin{aligned}
&(\wedge),(\vee) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}\\
&\mathsf{False} \wedge y = \mathsf{False}\\
&\mathsf{True} \ \wedge y = y \ ,\\
&\mathsf{False} \vee y = y\\
&\mathsf{True} \ \vee y = \mathsf{True} \ .
\end{aligned}
$$

運算子 $(\wedge)$與 $(\vee)$的定義同樣是各兩個子句，每個子句分別考慮其第一個參數的值。以 $x \wedge y$ 為例：如果 x 是 False, 不論 y 的值為何，$x \wedge y$ 都是 False；如果 x 是 True, $x \,\&\, y$ 的值和 y 相同。$(\vee)$ 的情況類似。

例 1.11. 以下函數判斷給定年份 y 是否為閏年。

$$
\begin{aligned}
&\mathit{leapyear} :: \mathsf{Int} \to \mathsf{Bool}\\
&\mathit{leapyear}\ y = (y \,`\mathit{mod}`\, 4 == 0) \wedge\\
&\qquad\qquad (y \,`\mathit{mod}`\, 100 \neq 0 \vee y \,`\mathit{mod}`\, 400 == 0) \ .
\end{aligned}
$$

我們來算算看 $\mathit{leapyear}\ 2016$。依照定義展開為

$$(2016 \,`\mathit{mod}`\, 4 == 0) \wedge (2016 \,`\mathit{mod}`\, 100 \neq 0 \vee 2016 \,`\mathit{mod}`\, 400 == 0) \ .$$

接下來該怎麼做呢？函數 $(\wedge)$ 的定義有兩個子句，我們得知道 $2016 \,`\mathit{mod}`\, 4 == 0$ 的值才能得知該歸約成哪個。因此只好先算 $2016 \,`\mathit{mod}`\, 4 == 0$，得到 True:

$$\mathsf{True} \wedge (2016 \,`\mathit{mod}`\, 100 \neq 0 \vee 2016 \,`\mathit{mod}`\, 400 == 0) \ ,$$

然後依照 $(\wedge)$ 的定義歸約為 $2016 \,`\mathit{mod}`\, 100 \neq 0 \vee 2016 \,`\mathit{mod}`\, 400 == 0$. 接下來也依此類推。

我們發現這是第 1.1節中所提及的被迫求值的例子：我們得先把參數算出，才知道接下來如何走。函數 *not*, $(\wedge)$, $(\vee)$ 定義成許多個子句，每個都分析其參數的可能外觀，據此決定該怎麼走。這種定義方式稱作樣式配對 ***(pattern matching)***：等號左手邊的 False, True 等等在此是樣式 (pattern)。使用這些函數時，例如 $x \wedge y$ 中，x 得先被算到可以和這些樣式配對上的程度，才能決定接下來的計算如何進行。

樣式配對也可用在不止一個參數上。例如，以下的運算元 $(==)$ 判斷兩個布林值是否相等。

$$
\begin{aligned}
&(==) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}\\
&\mathsf{False} == \mathsf{False} = \mathsf{True}\\
&\mathsf{False} == \mathsf{True} \ = \mathsf{False}\\
&\mathsf{True} == \mathsf{False} = \mathsf{False}\\
&\mathsf{True} == \mathsf{True} \ = \mathsf{True} \ .
\end{aligned}
$$

[9] 邏輯「且」又稱作合取 (conjunction)；邏輯「或」又稱作析取 (disjunction). 在 Haskell 中，「且」與「或」需分別寫成 (&&) 和 (||)。本書中採用數學與邏輯領域較常使用的 $(\wedge)$與 $(\vee)$.

讀者可能注意到我們用了同一個符號 $(==)$ 來表示整數與布林值的相等測試。請讀者暫且接受，相信 Haskell 有某些方式可得知任一個算式中的 $(==)$ 到底是什麼型別的相等。詳情 [todo: where?]

Haskell 中另有一個專用來做樣式配對的 **case** 算式。例如，$(\wedge)$ 也可寫成如下的形式：

$$
\begin{aligned}
&(\wedge) :: \mathsf{Bool} \to \mathsf{Bool} \\
&x \wedge y = \mathbf{case}\ x\ \mathbf{of} \\
&\qquad\qquad \mathsf{False} \to \mathsf{False} \\
&\qquad\qquad \mathsf{True}\ \to y\ .
\end{aligned}
$$

由於 **case** 是算式，如同 **let** 一樣可出現在其他算式中，也可巢狀出現。

習題 1.7 — 以 **case** 算式定義 *not*, $(\vee)$, 和 $(==)$.

習題 1.8 — 另一個定義 $(==) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$ 的方式是

$$x == y = (x \wedge y) \vee (not\ x \wedge not\ y)\ .$$

請將 $(x,y) := (\mathsf{False}, \mathsf{False})$, $(x,y) := (\mathsf{False}, \mathsf{True})$ 等四種可能分別代入化簡，看看是否和本節之前的 $(==)$ 定義相同。

1.6.2 字元

我們可把「字元」這個型別想成一個很長的 **data** 宣告：

$$\mathbf{data}\ \mathsf{Char} = \texttt{'a'} \mid \texttt{'b'} \mid \ldots \mid \texttt{'z'} \mid \texttt{'A'} \mid \texttt{'B'} \ldots .$$

其中包括所有字母、符號、空白... 目前的 Haskell 甚至有處理 Unicode 字元的能力。但無論如何，Char 之中的字元數目是有限的。我們可用樣式配對定義字元上的函數。注意：字元以單引號括起來。

我們也可假設字元是有順序的，每個字元對應到一個內碼。關於 Char 的常用函數中，*ord* 將字元的內碼找出，*chr* 則將內碼轉為字元：

$$
\begin{aligned}
&ord :: \mathsf{Char} \to \mathsf{Int}\ , \\
&chr :: \mathsf{Int} \to \mathsf{Char}\ .
\end{aligned}
$$

例 1.12. 下列函數 *isUpper* 判斷一個字元是否為大寫英文字母；*toLower* 則將大寫字母轉成小寫字母，若輸入並非大寫字母則不予以變動。

$$
\begin{aligned}
&isUpper :: \mathsf{Char} \to \mathsf{Bool} \\
&isUpper\ c = \mathbf{let}\ x = ord\ c\ \mathbf{in}\ ord\ \texttt{'A'} \leqslant x \wedge x \leqslant ord\ \texttt{'Z'}\ , \\
&toLower :: \mathsf{Char} \to \mathsf{Char} \\
&toLower\ c \mid isUpper\ c = chr\ (ord\ c - ord\ \texttt{'A'} + ord\ \texttt{'a'}) \\
&\qquad\qquad \mid otherwise = c\ .
\end{aligned}
$$

1.6.3 序對

數學上，將兩個值（如 3 和 'a'）放在一起，就成了一個有序對 (*ordered pair*)，可寫成 $(3,\texttt{'a'})$。之所以稱作「有序」對，因為其中兩個元素的順序是不可忽略的 — $(3,\texttt{'a'})$ 與 $(\texttt{'a'},3)$ 是不同的有序對。另一個常見譯名是「數對」。由於我們處理的不只是數字，本書將之簡稱為「序對」。

給兩個集合 A 和 B, 從 A 之中任取一元素 x，從 B 之中也任取一元素 y，兩者的序對 (x,y) 形成的集合稱作 A 和 B 的笛卡兒積 (*Cartesian product*)，寫成 $\mathsf{A}\times\mathsf{B}$:

$$\mathsf{A}\times\mathsf{B}=\{(x,y)\mid x\in\mathsf{A},y\in\mathsf{B}\}\ .$$

Haskell 之中也有類似的構造。給定型別 a 與 b, 它們的序對的型別是 $(a\times b)$. [10] 我們可以想像 Haskell 有這麼一個型別定義：

$$\textbf{data}\ (a\times b)=(a,b)\ .$$

以口語說的話，$(a\times b)$ 是一個新型別，而具有此型別的值若有範式，必定是 (x,y) 的形式，其中 x 的型別是 a, y 的型別是 b. [11] 序對的建構元寫成 $(,)$，型別為 $a\to b\to(a\times b)$. 例如 $(,)\ 4\ \texttt{'b'}=(4,\texttt{'b'})$.

兩個常用的函數 *fst* 與 *snd* 分別取出序對中的第一個和第二個元素：

$$\begin{aligned} &fst::(a\times b)\to a && snd::(a\times b)\to b\\ &fst\ (x,y)=x\ , && snd\ (x,y)=y\ . \end{aligned}$$

函數 *fst* 與 *snd* 的定義方式也是樣式配對：輸入值必須先計算成 (x,y) 的形式。

例 1.13. 以下是一些序對與其相關函數的例子。

- $(3,\texttt{'a'})$ 是一個型別為 $(\mathsf{Int}\times\mathsf{Char})$ 的序對。
- $fst\ (3,\texttt{'a'})=3$, $snd\ (3,\texttt{'a'})=\texttt{'a'}$
- 函數 *swap* 將序對中的元素調換：

$$\begin{aligned} &swap::(a\times b)\to(b\times a)\\ &swap\ (x,y)=(y,x)\ . \end{aligned}$$

另一個定義方式是 $swap\ p=(snd\ p,fst\ p)$. 但這兩個定義並不盡然相同。詳見第 *1.7*節。

序對也可以巢狀構成。例如 $((\mathsf{True},3),\texttt{'c'})$ 是一個型別為 $((\mathsf{Bool}\times\mathsf{Int})\times\mathsf{Char})$ 的序對，而 $snd\ (fst\ ((\mathsf{True},3),\texttt{'c'}))=3$. 在 Haskell 之中，$((a\times b)\times c)$ 與 $(a\times(b\times c))$ 被視為不同的型別，但他們是同構的 — 我們可定義一對函數在這兩個型別之間作轉換：

$$\begin{aligned} &assocr::((a\times b)\times c)\to(a\times(b\times c))\\ &assocr\ ((x,y),z)=(x,(y,z))\ , \end{aligned}$$

[10] 然而，由於「程式可能不終止」這個因素作怪，$a\times b$ 的元素並不僅是 a 與 b（如果視做集合）的笛卡兒積。詳見 [todo: where?]

[11] 其實這個定義並不符合 Haskell 的語法，因此只是方便理解的想像。另，型別 $(a\times b)$ 在 Haskell 中寫成 (a,b). 我的經驗中，讓型別與值的語法太接近，反易造成困惑。

同構

兩個集合 A與 B 同構 (*isomorphic*)，意思是 A 之中的 每個元素都唯一地對應到 B 之中的一個元素，反之亦然。

一個形式定義是：A與 B同構意謂我們能找到兩個全 (total) 函數 $to :: A \to B$ 和 $from :: B \to A$, 滿足

$$\begin{aligned} from \cdot to &= id \ , \\ to \cdot from &= id \ . \end{aligned}$$

此處的兩個 id 型別依序分別為 $A \to A$ 和 $B \to B$。將定義展開，也就是說，對所有 $x :: A$，$from\ (to\ x) = x$; 對所有 $y :: B$, $to\ (from\ y) = y$. 這個定義迫使對每個 x 都存在一個唯一的 $to\ x$, 反之亦然。

我們已有兩個例子：$((a \times b) \times c)$ 與 $(a \times (b \times c))$ 同構，此外，$(a \times b)$ 與 $(b \times a)$ 也同構，因為 $swap \cdot swap = id$.

如果集合 A與 B同構，不僅 A 之中的每個元素都有個在 B 之中相對的元素，給任一個定義在 A 之上的函數 f, 我們必可構造出一個 B 之上的函數，具有和 f 相同的性質。即使 A與 B並不真正相等，我們也可把它們視為基本上沒有差別的。在許多無法談「相等」的領域中，同構是和「相等」地位一樣的觀念。

$$\begin{aligned} &assocl :: (a \times (b \times c)) \to ((a \times b) \times c) \\ &assocl\ (x, (y, z)) = ((x, y), z) \ , \end{aligned}$$

並且滿足 $assocr \cdot assocl = id$, 和 $assocl \cdot assocr = id$.

習題 1.9 — 試試看不用樣式配對，而以 fst 和 snd 定義 $assocl$ 和 $assocr$:

$$\begin{aligned} &assocl\ p = \ldots \\ &assocr\ p = \ldots \end{aligned}$$

另外可一提的是，Haskell 允許我們在 λ 算式中做樣式配對。例如 fst 的另一種寫法是：

$$fst = \lambda (x, y) \to x \ .$$

Haskell 另有提供更多個元素形成的有序組，例如 (True, 3, 'c') 是一個型別為 (Bool × Int × Char) 的值。但本書暫時不使用他們。

分裂與積 在我們將介紹的程式設計風格中，以下兩個產生序對的運算子相當好用。第一個運算子利用兩個函數產生一個序對：

$$\begin{aligned} &\langle \cdot, \cdot \rangle :: (a \to b) \to (a \to c) \to a \to (b \times c) \\ &\langle f, g \rangle\ x = (f\ x, g\ x) \ . \end{aligned}$$

給定兩個函數 $f :: a \to b$ 和 $g :: a \to c$, $\langle f, g \rangle :: a \to (b \times c)$ 是一個新函數，將 f 和 g 的結果收集在一個序對中。我們借用範疇論的詞彙，將此稱作分裂 (*split*) — $\langle f, g \rangle$ 可讀成「f 與 g 的分裂」。

如果我們已經有了一個序對，我們可用 $(f \times g)$ 算出一個新序對：

$$(\times) :: (a \to b) \to (c \to d) \to (a \times c) \to (b \times d)$$
$$(f \times g)\ (x,y) = (f\ x, g\ x)\ .$$

函數 $(f \times g)$ 將 f 和 g 分別作用在序對 (x,y) 的兩個元素上。這個操作稱作「f 和 g 的乘績 (product)」，同樣是借用範疇論的詞彙。

目前 Haskell 的標準階層函式庫將分裂與乘積收錄在 `Control.Arrow` 中，$\langle f,g \rangle$ 寫作：`f &&& g`, $(f \times g)$ 則寫作 `f *** g`.

Currying 與 Uncurrying 如前所述，Haskell 的每個函數都只拿一個參數。拿多個參數的函數可以傳回函數的函數來模擬，稱作 currying. 有了序對之後，另一種模擬多參數的方式是把參數都包到一個序對中。例如，型別為 $(a \times b) \to c)$ 的函數可視為拿了兩個型別為 a 與 b 的參數。

函數 *curry* 與 *uncurry* 幫助我們在這兩種表示法之間轉換 — *curry* 將拿序對的函數轉換成 curried 函數，*uncurry* 則讓 curried 函數改拿序對當作參數：

$$curry :: ((a \times b) \to c) \to (a \to b \to c)$$
$$curry\ f\ x\ y = f\ (x,y)\ ,$$
$$uncurry :: (a \to b \to c) \to ((a \times b) \to c)$$
$$uncurry\ f\ (x,y) = f\ x\ y\ .$$

例：如果 $(==)$ 的型別為 $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool}$, $uncurry\ (==)$ 的型別為 $(\mathsf{Int} \times \mathsf{Int}) \to \mathsf{Bool}$。後者檢查一個序對中的兩個值是否相等（例：$uncurry\ (==)\ (3,3) = \mathsf{True}$）。

習題 1.10 — 事實上，*curry* 與 *uncurry* 的存在證明了 $(a \times b) \to c$ 與 $a \to b \to c$ 是同構的。試證明 $curry \cdot uncurry = id$, 以及 $uncurry \cdot curry = id$.

習題 1.11 — 請說明 $map\ (uncurry\ (\$))$ 的型別與功能。關於 $(\$)$ 請參考第 30頁。

1.7 弱首範式

我們現在可把第 1.1節中提及的求值與範式談得更仔細些。第一次閱讀的讀者可把本節跳過。回顧 *fst* 使用樣式配對的定義 $fst\ (x,y) = x$. 假設我們把 *swap* 定義如下：

$$swap\ p = (snd\ p, fst\ p)\ .$$

考慮 $fst\ (swap\ (3, \texttt{'a'}))$ 該怎麼求值：

$$\begin{aligned} & fst\ (swap\ (3, \texttt{'a'})) \\ = \quad & \{ swap \text{ 的定義} \} \\ & fst\ (snd\ (3, \texttt{'a'}), fst\ (3, \texttt{'a'})) \\ = \quad & \{ fst \text{ 的定義} \} \\ & snd\ (3, \texttt{'a'}) \end{aligned}$$

$$
\begin{aligned}
&= \quad \{ snd \text{ 的定義} \} \\
&\quad \texttt{'a'} \ .
\end{aligned}
$$

在第一步中，由於 fst 使用樣式配對，我們得先把 $swap\ (3,\texttt{'a'})$ 算出來。若把 $swap\ (3,\texttt{'a'})$ 算到底，得到的範式是 $(\texttt{'a'},3)$. 但如第一步中所顯示，如果目的只是為了配對 (x,y) 這個樣式，我們並不需要把 $swap\ (3,\texttt{'a'})$ 算完，只需算到 $(snd\ (3,\texttt{'a'}),fst\ (3,\texttt{'a'}))$ 即可 — x 可對應到 $snd\ (3,\texttt{'a'})$, y 可對應到 $fst\ (3,\texttt{'a'})$, fst 的計算便可以進行。在下一步中，子算式 $fst\ (3,\texttt{'a'})$ 便被丟棄了，並沒有必要算出來。

做樣式配對時，Haskell 只會把算式歸約到剛好足以與樣式配對上的程度。當樣式的深度只有一層（如 (x,y)）時，與之配對的式子會被歸約成一種稱作弱首範式 (*weak head normal form*)的形式。弱首範式有其嚴格定義，但本書讀者只需知道：算式會被歸約到露出最外面的建構元，（在此例中是被歸約成 $(\_,\_)$ 的形式），然後便停下來。

例 1.14. 回顧 $swap$ 的兩種定義方式，分別命名為

$$
\begin{aligned}
swap_1\ (x,y) &= (y,x) \ , \\
swap_2\ p \quad &= (snd\ p, fst\ p) \ ,
\end{aligned}
$$

若考慮不終止的參數，兩者的行為並不盡然相同。定義 $three\ (x,y) = 3$, 並假設 $\bot$ 是一個沒有範式的式子 — 一旦開始對 $\bot$ 求值，便停不下來。試計算 $three\ (swap_1\ \bot)$ 和 $three\ (swap_2\ \bot)$.

答. 由於 $three$ 使用樣式配對，計算 $three\ (swap_1\ \bot)$ 時得把 $swap_1\ \bot$ 歸約成弱首範式。同理，計算 $swap_1\ \bot$ 時，第一步便是先試圖把 $\bot$ 歸約成弱首範式，然後便停不下來了。

至於 $three\ (swap_2\ \bot)$ 則可如下地求值：

$$
\begin{aligned}
&\quad three\ (swap_2\ \bot) \\
&= \quad \{ swap_2 \text{ 之定義} \} \\
&\quad three\ (snd\ \bot, fst\ \bot) \\
&= \quad \{ three \text{ 之定義} \} \\
&\quad 3 \ .
\end{aligned}
$$

第一步中，$swap_2\ \bot$ 則依照範式順序求值的原則展開為 $(snd\ \bot, fst\ \bot)$ — 這是一個序對，只是該序對含有兩個沒有範式的元素。該序對可對應到樣式 (x,y), 因此整個式子歸約為 3.

附帶一提，$three\ \bot$ 是一個不停止的計算。 □

1.8 串列

一個串列 (list) 抽象說來便是將零個或多個值放在一起變成一串。串列是函數語言傳統上的重要資料結構：早期的函數語言 LISP 便是 LISt Processing 的縮寫。Haskell 中的串列多了一個限制：串列中的每個元素必須有同樣的型別。

(:) 與 (::)

大部分有型別的函數語言（如 ML, Agda 等）之中，(:) 表示型別關係，(::) 則是串列的建構元。Haskell 的前身之一是 David A. Turner 的語言 Miranda. 在其 Hindley-Milner 型別系統中，Miranda 使用者幾乎不需寫出程式的型別 — 型別可由電腦自動推導。而串列是重要資料結構。把兩個符號調換過來，使常用的符號較短一些，似乎是合理的設計。

Haskell 繼承了 Miranda 的語法。然而，後來 Haskell 的型別發展得越來越複雜，使用者偶爾需要寫出型別來幫助編譯器。即使型別簡單，程式語言界也漸漸覺得將函數的型別寫出是好習慣。而串列建構元的使用量並不見得比型別關係多。但此時想改符號也為時已晚了。

本書中將「元素型別均為 a 的串列」的型別寫成 List a. [12] Hasekell 以中括號表示串列，其中的元素以逗號分隔。例如，$[1,2,3,4]$ 是一個型別為 List Int 的串列，其中有四個元素；$[\mathsf{True},\mathsf{False},\mathsf{True}]$ 是一個型別為 List Bool 的串列，有三個元素。至於 $[\,]$ 則是沒有元素的空串列（通常唸做 “nil”），其最通用的型別為 List a，其中 a 可以是任何型別。

串列的元素也可以是串列。例如 $[[1,2,3],[],[4,5]]$ 的型別是 List (List Int), 含有三個元素，分別為 $[1,2,3]$, $[\,]$, 和 $[4,5]$.

事實上，上述的寫法只是語法糖。我們可想像 Haskell 有這樣的型別定義：

$$\textbf{data}\ \mathrm{List}\ a = [\,] \mid a : \mathrm{List}\ a \ .$$

意謂一個元素型別為 a 的串列只有兩種可能構成方式：可能是空串列 $[\,]$，也可能是一個元素 (a) 接上另一個串列 (List a)。後者的情況中，元素和串列之間用符號 (:) 銜接。

符號 (:) 唸作 “cons”, 為「建構 (construct)」的字首。其型別為 $a \to \mathrm{List}\ a \to \mathrm{List}\ a$ — 它總是將一個型別為 a 的元素接到一個 List a 之上，造出另一個 List a. 上述的 $[1,2,3,4]$ 其實是 $1:(2:(3:(4:[\,])))$ 的簡寫：由空串列 $[\,]$ 開始，將元素一個個接上去。為了方便，Haskell 將 (:) 運算元視做右相依的，因此我們可將括號省去，寫成 $1:2:3:4:[\,]$。 無論如何，這樣的串列表示法是偏一邊的 — 元素總是從左邊放入，最左邊的元素也最容易取出。如果一個串列不是空的，其最左邊的元素稱作該串列的頭 (*head*)，剩下的元素稱作其尾 (*tail*)。例如，$[1,2,3,4]$ 的頭是 1, 尾是 $[2,3,4]$.

Haskell 中將字串當作字元形成的串列。標準函式庫中這麼定義著：

$$\textbf{type}\ \mathrm{String} = \mathrm{List}\ \mathrm{Char} \ .$$

意謂 String 就是 List Char。在 Haskell 中，**data** 用於定義新型別，而 **type** 並不產生一個新的型別，只是給現有的型別一個較方便或更顯出當下意圖的名字。此外，Haskell 另提供一個語法糖，用雙引號表達字串。因此，`"fun"` 是 `['f','u','n']` 的簡寫，後者又是 `'f':'u':'n':[]` 的簡寫。

本節接下來將介紹許多與串列相關的內建函數。

[12] Haskell 中的寫法是 $[a]$. 同樣地，在我的教學經驗中，將中括號同時使用在值與型別上造成不少誤解。例如學生可能認為 $[1,2]$ 的型別是 $[\mathrm{Int},\mathrm{Int}]$ — 其實應該是 $[\mathrm{Int}]$.

1.8.1 串列解構

我們先從拆解串列的函數開始。函數 *head* 和 *tail* 分別取出一個串列的頭和尾：

$$\begin{aligned} &head :: \text{List } a \to a && tail :: \text{List } a \to \text{List } a \\ &head\ (x : xs) = x\ , && tail\ (x : xs) = xs\ . \end{aligned}$$

注意其型別：*head* 傳回一個元素，*tail* 則傳回一個串列。例：*head* `"fun"` 和 *tail* `"fun"` 分別是字元 `'f'` 和字串 `"un"`. 函數 *head* 和 *tail* 都可用樣式配對定義，但此處的樣式並不完整，尚缺 $[\,]$ 的情況。如果將空串列送給 *head* 或 *tail*，則會出現執行時錯誤。因此，*head* 和 *tail* 都是部分函數 (*partial functions*) — 它們只將某些值（非空的串列）對應到輸出，某些值（空串列）則沒有。

函數 *null* 判斷一個串列是否為空串列。它也可用樣式配對定義如下：

$$\begin{aligned} &null :: \text{List } a \to \text{Bool} \\ &null\ [\,] \quad\ = \text{True} \\ &null\ (x : xs) = \text{False}\ . \end{aligned}$$

本書依循 Bird [1998] 中的變數命名習慣，將型別為串列的變數以 *s* 做結尾，例如 *xs*, *ys* 等等。至於「元素為串列的串列」則命名為 *xss*, *yss* 等等。但這只是為方便理解而設計的習慣。Haskell 本身並無此規定。

除了 *head* 與 *tail*，也有另一組函數 *last* 與 *init* 分別取出一個串列最右邊的元素，以及剩下的串列：

$$\begin{aligned} &last :: \text{List } a \to a\ , \\ &init :: \text{List } a \to \text{List } a\ . \end{aligned}$$

例：*last* `"fun"` 與 *init* `"fun"` 分別為字元 `'n'` 與字串 `"fu"`. 但 *last* 與 *init* 的定義比起 *head* 與 *tail* 來得複雜：記得我們的串列表示法是偏向一邊的，從左邊存取元素容易，從右邊存取元素則較麻煩。我們會在 [todo: where] 之中談到 *last* 與 *init* 的定義。

1.8.2 串列生成

第 1.8.1節中的函數均將串列拆開。本節之中我們來看一些生成串列的方法。如果元素的型別是有順序的（例如 Int, Char等型別），Haskell 提供了一個方便我們依序生成串列的語法。以例子說明：

例 1.15. 以下為 *Haskell* 的列舉語法的一些例子：

- $[0..10]$ 可展開為 $[0,1,2,3,4,5,6,7,8,9,10]$.
- 可用頭兩個元素來指定間隔量。例如 $[0,3..10] = [0,3,6,9]$. 注意該串列的元素不超過右界 10.
- 在 $[10..0]$ 之中，10 一開始就超過了右界 0, 因此得到 $[\,]$. 如果想要產生由 10 倒數到 0 的串列，可這樣指定間隔：$[10,9..0]$.
- 字元也是有順序的，因此 [`'a'..'z'`] 可展開為含所有英文小寫字母的串列。
- 至於沒有右界的 $[0..]$ 則會展開為含 $[0,1,2,3...]$ 的無限長串列。

函數 $iterate :: (a \to a) \to a \to \text{List } a$ 用於產生無限長的串列：$iterate\ f\ x$ 可展開為 $[x, f\ x, f\ (f\ x), f\ (f\ (f\ x))...]$.

例 1.16. 一些 *iterate* 的例子：

- $iterate\ (1+)\ 0$ 展開為 $[0,1,2,3...]$. 其實 $[n..]$ 可視為 $iterate\ (1+)\ n$ 的簡寫。
- 在例 *1.24*中我們會看到 $[m..n]$ 也可用 *iterate* 與其他函數做出。
- $iterate\ not\ \text{False}$ 可得到無窮串列 $[\text{False}, \text{True}, \text{False}...]$.

數學中描述集合時常使用一種稱作集合建構式 (set comprehension) 的語法。例如，$\{x \times x \mid x \in S, odd\ x\}$ 表示收集所有 $x \times x$ 形成的集合，其中 x 由集合 S 中取出，並且必須為奇數。Haskell 將類似的語法用在串列上。同樣以例子說明：

例 1.17. 串列建構式 *(list comprehension)* 的例子：

- $[x \mid x \leftarrow [0..9]]$ 表示「從 $[0..9]$ 之中取出 x, 並收集 x」, 可展開為 $[0,1,2,3,4,5,6,7,8,9]$.
- $[x \times x \mid x \leftarrow [0..10]]$ 的 x 來源和之前相同，但收集的是 $x \times x$，得到 $[0,1,4,9,25,36,49,64,81]$.
- $[(x,y) \mid x \leftarrow [0..2], y \leftarrow \texttt{"abc"}]$ 展開得到 $[(0, \texttt{'a'}), (0, \texttt{'b'}), (0, \texttt{'c'}), (1, \texttt{'a'}), (1, \texttt{'b'}), (1, \texttt{'c'}), (2, \texttt{'a'}), (2, \texttt{'b'}), (2, \texttt{'c'})]$. 注意序對出現的順序：先固定 x，將 y 跑過一遍，再換成下一個 x.
- $[x \times x \mid x \leftarrow [0..10], odd\ x]$ 從 $[0..10]$ 之中取出 x，但只挑出滿足 $odd\ x$ 的那些，得到 $[1,9,25,49,81]$.

例 1.18. 以下算式的值分別為何?

1. $[(a,b) \mid a \leftarrow [1..3], b \leftarrow [1..2]]$.
2. $[(a,b) \mid b \leftarrow [1..2], a \leftarrow [1..3]]$.
3. $[(i,j) \mid i \leftarrow [1..4], j \leftarrow [(i+1)..4]]$. 這是一個有了 $i \leftarrow ..$ 之後，i 即可在右方被使用的例子。
4. $[(i,j) \mid i \leftarrow [1..4], even\ i, j \leftarrow [(i+1)..4], odd\ j]$.
5. $[\texttt{'a'} \mid i \leftarrow [0..10]]$. 這個例子顯示 i 並不一定非得出現在被收集項目中。

答. 分別展開如下：

1. $[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]$.
2. $[(1,1),(2,1),(3,1),(1,2),(2,2),(3,2)]$.
3. $[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]$.
4. $[(2,3)]$.
5. `"aaaaaaaaaa"`.

□

串列建構式在寫程式時相當好用，但它也僅是個語法糖 – 所有的串列建構式都可轉換為後面的章節將介紹的 *map*, *concat*, *filter* 等函數的組合。

1.8.3 串列上的種種組件函數

我們將在本節介紹大量與串列有關的函數。它們常被稱做組件 (*combinators*)函數。每一個組件都負責一項單一、但具通用性而容易重用的功能。它們常用來彼此結合以組出更大的程式。

介紹這些函數有兩個原因。首先，它們可用來組出許多有趣的程式。我們將一邊逐一介紹這些函數，一邊以例子示範，同時也逐漸帶出本章鼓勵的一種特殊編程風格。另一個原因是在日後的章節中我們也都將以這些函數作為例子，討論並證明關於它們的性質。

第一次閱讀的讀者可能訝異：這麼多函數，怎麼記得住？事實上，這些組件大都僅是把常見的編程模式具體地以形式方式表達出來。辨識出這些模式後，不僅會發現它們其實很熟悉，對於我們日後了解其他程式也有助益。

長度 函數 $length :: \mathsf{List}\ a \to \mathsf{Int}$ 計算串列的長度。空串列 $[]$ 的長度為 0. 例：$length\ \texttt{"function"} = 8$.

索引 函數 $(!!)$ 的型別為 $\mathsf{List}\ a \to \mathsf{Int} \to a$. 給定串列 xs 和整數 i, 如果 $0 \leqslant i < length\ xs$, $xs !! i$ 為 xs 中的第 i 個元素，但由 0 起算。例 $\texttt{"function"} !! 0 = \texttt{'f'}$, $\texttt{"function"} !! 3 = \texttt{'c'}$. 如果 $i > length\ xs$, 則會成為執行期錯誤。注意：如果 $length\ xs = n$，其中的元素編號分別為 $0, 1 .. n-1$.

在指令式語言中，索引是處理陣列常用的基本操作。處理陣列的常見模式是用一個變數 i指向目前正被處理的元素，將 $a\ [i]$的值讀出或覆寫，然後更新 i的值。但由接下來的許多範例中，讀者會發現本章盡量避免這種做法。也因此 $(!!)$ 在本章中使用的機會不多。

連接 函數 $(+\!\!+) :: \mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a$ 將兩個串列相接。例：$[1,2,3] +\!\!+ [4,5] = [1,2,3,4,5]$.

函數 $(+\!\!+)$ 和 $(:)$ 似乎都是把串列接上東西。兩者有什麼不同呢？答案是：$(:) :: a \to \mathsf{List}\ a \to \mathsf{List}\ a$ 永遠把一個元素接到串列的左邊，而 $(+\!\!+)$ 把兩個串列接在一起，兩個串列都有可能含有零個或多個元素。例：$[] +\!\!+ [4,5] = [4,5]$. 事實上，$(:)$ 是比 $(+\!\!+)$ 更基礎的操作。在第 2.4.1節中，我們會看到 $(+\!\!+)$ 是用 $(:)$ 定義而成的。

另一個關於連接的函數是 $concat :: \mathsf{List}\ (\mathsf{List}\ a) \to \mathsf{List}\ a$：它以一個元素都是串列的串列作為輸入，將其中的串列接在一起。例：$concat\ [[1,2,3],[],[4],[5,6]] = [1,2,3,4,5,6]$. 它和 $(+\!\!+)$ 的不同之處在哪呢？顯然，$(+\!\!+)$ 總把兩個串列接在一起，而 $concat$ 的參數中可含有零個或多個串列。在第 2.4.1節中，我們會看到 $concat$ 是用 $(+\!\!+)$ 定義而成的。

取與丟 $take$ 的型別為 $\mathsf{Int} \to \mathsf{List}\ a \to \mathsf{List}\ a$. $take\ n\ xs$ 取 xs 的前 n 個元素。若 xs 的長度不到 n, $take\ n\ xs$ 能拿幾個就拿幾個. 例：$take\ 3\ \texttt{"function"} = \texttt{"fun"}$, $take\ 5\ \texttt{"ox"} = \texttt{"ox"}$.

相對地，$drop\ n\ xs$ 丟掉 xs 的前 n 個元素。若 xs 的長度不到 n, $drop\ n\ xs$ 能丟幾個就拿幾個. 例：$drop\ 3\ \texttt{"function"} = \texttt{"ction"}$, $take\ 5\ \texttt{"ox"} = \texttt{""}$. 函數 $drop$ 的型別也是 $\mathsf{Int} \to \mathsf{List}\ a \to \mathsf{List}\ a$.

函數 $take$ 和 $drop$ 顯然有些關聯，但它們的關聯該怎麼具體地寫下來呢？一個可能是：對所有的 n 和 xs,

$$take\ n\ xs +\!\!+ drop\ n\ xs = xs \quad .$$

乍看之下似乎言之成理。但這個性質真的成立嗎？我們將在第 2章中討論到。

映射 $map :: (a \to b) \to \mathsf{List}\, a \to \mathsf{List}\, b$ 是串列上一個很重要的高階函數：$map\, f\, xs$ 將 f 作用在 xs 的每一個元素上。例：

$$map\ square\ [1,2,3,4] = [1,4,9,16]\ ,$$
$$map\ (1+)\ [2,3,4] = [3,4,5]\ .$$

回憶我們之前關於高階函數的討論，另一個理解方式是：map 是一個處理函數的操作。給一個「將 a 變成 b」的函數 $f :: a \to b$，map 將這個函數提升到串列的層次，得到一個「將 $\mathsf{List}\, a$ 變成 $\mathsf{List}\, b$」的函數 $map\, f :: \mathsf{List}\, a \to \mathsf{List}\, b$.

例 1.19. 如果一個串列 xs 可分解為 $ys +\!\!+ zs$, 我們說 ys 是 xs 的一個前段 (prefix), zs 則是 xs 的一個後段 (suffix). 例如，串列 $[1,2,3]$ 的前段包括 $[]$, $[1]$, $[1,2]$, 與 $[1,2,3]$（注意：$[]$是一個前段，$[1,2,3]$ 本身也是），後段則包括 $[1,2,3]$, $[2,3]$, $[3]$, 與 $[]$。

試定義函數 $inits :: \mathsf{List}\, a \to \mathsf{List}\, (\mathsf{List}\, a)$, 計算輸入串列的所有前段。[13] **提示**：目前我們可以用 map, $take$ 和其他函數組出 $inits$. 在第 2.6節中將會介紹另一個做法。

答. 一種使用 map 和 $take$ 的可能作法如下：

$$\begin{aligned} &inits :: \mathsf{List}\, a \to \mathsf{List}\, (\mathsf{List}\, a) \\ &inits\ xs = map\ (\lambda n \to take\ n\ xs)\ [0\,..\,length\ xs]\ . \end{aligned}$$

或著也可用串列建構式寫成 $[take\ n\ xs \mid n \leftarrow [0\,..\,length\ xs]]$. 讀者可能已發現：$[f\ x \mid x \leftarrow xs]$ 就是 $map\, f\, xs$. □

習題 1.12 — 定義函數 $tails :: \mathsf{List}\, a \to \mathsf{List}\, (\mathsf{List}\, a)$, 計算輸入串列的所有後段。

過濾 一個型別為 $a \to \mathsf{Bool}$ 的函數稱作一個「述語」(predicate). 給定述語 p, $filter\ p\ xs$ 將 xs 之中滿足 p 的元素挑出。函數 $filter$ 的型別為 $(a \to \mathsf{Bool}) \to \mathsf{List}\, a \to \mathsf{List}\, a$。例：$filter\ even\ [2,5,1,7,6] = [2,6]$.

例 1.20. 該怎麼得知一個字串中大寫字母的個數？將大寫字母過濾出來，計算所得串列的長度即可。如下所示：

$$\begin{aligned} &numUpper :: \mathsf{String} \to \mathsf{Int} \\ &numUpper = length \cdot filter\ isUpper\ . \end{aligned}$$

例 1.21. 下列算式求出 0^2 到 50^2 的平方數（能寫成 n^2 的數字）中，結尾為 25 的數字。

$$filter\ ((== 25) \cdot (`mod`100))\ (map\ square\ [0\,..\,50])\ .$$

[13] 請注意該函數的名字是 $inits$, 和之前介紹過的 $init$ 不同。這是 Haskell 函式庫中使用的命名。

歸約後得到 $[25,225,625,1225,2025]$. 其中 $(==25)\cdot(`mod`100)$ 部分使用了第 25頁 中提到的語法。如果覺得不習慣，也可用 λ 算式寫成：

$$filter\ (\lambda n \to n\ `mod`\ 100 == 25)\ (map\ square\ [0..50])\ .$$

例 1.22. 接續上例。另一個可能寫法是先過濾出「平方之後結尾為 25」的數字，再算這些數字的平方：

$$map\ square\ (filter\ ((==25)\cdot(`mod`100)\cdot square)\ [0..50])\ .$$

這個算式也歸約出一樣的結果：$[25,225,625,1225,2025]$。

稍微推廣一些，這個例子暗示我們 $filter\ p\cdot map\ f$ 和 $map\ f\cdot filter\ (p\cdot f)$ 似乎是等價的。但確實如此嗎？我們也將在第 2章 中討論。

例 1.23. 接續上例。如果我們不僅希望找到結尾為 25 的平方數，也希望知道它們是什麼數字的平方，一種寫法如下：

$$filter\ ((==25)\cdot(`mod`100)\cdot snd)\ (map\ \langle id,square\rangle\ [0..50])\ .$$

我們用 $map\ \langle id,square\rangle$ 將每個數字與他們的平方放在一個序對中，得到 $[(0,0),(1,1),(2,4),(3,9)...]$. 而 $filter$ 的述語多了一個 snd, 表示我們只要那些「第二個元素符合條件」的序對。上式化簡後可得到 $[(5,25),(15,225),(25,625),(35,1225),(45,2025)]$. 運算元 $\langle\cdot,\cdot\rangle$ 的定義詳見第 36頁。

述語 $(==25)\cdot(`mod`100)\cdot snd$ 可以展開為 $(\lambda(i,n)\to n\ `mod`\ 100 == 25)$.

取、丟、與過濾 函數 $takeWhile$, $dropWhile$ 和 $filter$ 有一樣的型別。

$$takeWhile :: (a\to \mathsf{Bool})\to \mathsf{List}\ a\to \mathsf{List}\ a\ ,$$
$$dropWhile :: (a\to \mathsf{Bool})\to \mathsf{List}\ a\to \mathsf{List}\ a\ .$$

它們之間的差異也許用例子解釋得最清楚：

$$filter\ \ even\ [6,2,4,1,7,8,2] = [6,2,4,8,2]\ ,$$
$$takeWhile\ even\ [6,2,4,1,7,8,2] = [6,2,4]\ ,$$
$$dropWhile\ even\ [6,2,4,1,7,8,2] = [1,7,8,2]\ .$$

$filter\ p$ 挑出所有滿足 p 的元素；$takeWhile\ p$ 由左往右逐一取出元素，直到遇上第一個不滿足 p 的元素，並將剩下的串列丟棄；$dropWhile\ p$ 則與 $takeWhile\ p$ 相對，將元素丟棄，直到遇上第一個不滿足 p 的元素。直覺上，後兩者似乎也應該滿足 $takeWhile\ p\ xs +\!\!+ dropWhile\ p\ xs = xs$, 但這仍尚待驗證。

例 1.24. 給定整數 m 與 n, $[m..n]$ 可視為 $takeWhile\ (\leqslant n)\ (iterate\ (1+)\ m)$ 的簡寫。

例 1.25. 讀者也許覺得 $takeWhile$ 或 $dropWhile$ 似乎和迴圈有密切關係。確實，利用 $iterate$ 與 $dropWhile$，我們可定義出類似 $while$ 迴圈的操作：

$$until :: (a\to \mathsf{Bool})\to (a\to a)\to a\to a$$
$$until\ p\ f = head\cdot dropWhile\ (not\cdot p)\cdot iterate\ f$$

until p f x 由 *x* 算出 *f x*, 由 *f x* 算出 *f (f x)* ... 直到 *p (f (f ..x))* 成立為止。例: *until ((>50) · square) (1+) 0* 得到 8, 因為 $8^2 = 64$, 是第一個平方大於 50 的非負整數。由於惰性求值, *iterate f* 在意義上雖然是個無窮串列, 但只會被執行到 *dropWhile (not · p)* 擷取的長度為止。

下述函數則實作了用輾轉相減法求最大公因數的古典演算法。函數 *minus* 不斷將大數減去小數, 直到兩數相等為止:

```
gcd :: (Int × Int) → Int
gcd = fst · until (uncurry (==)) minus  ,
  where minus (x,y) | x > y = (y,x−y)
                    | x < y = (y−x,x)  .
```

關於 *uncurry* 詳見第 *37*頁。

習題 1.13 — 試定義一個函數 *squaresUpTo* :: Int → List Int, 使得 *squaresUpTo n* 傳回所有不大於 *n* 的平方數。例：*squaresUpTo* 10 = [1,4,9], *squaresUpTo* (−1) = [].

拉鍊 函數 *zip* :: List *a* → List *b* → List (*a* × *b*) 的作用可由下述例子示範：

```
zip [1,2,3] "abc" = [(1,'a'),(2,'b'),(3,'c')] ,
zip [1,2]   "abc" = [(1,'a'),(2,'b')]         ,
zip [1,2,3] "ab"  = [(1,'a'),(2,'b')]         ,
zip [1..]   "abc" = [(1,'a'),(2,'b'),(3,'c')] ,
zip [1..]   [2..] = [(1,2),(2,3),(3,4)..]     ,
```

zip xs ys 將串列 *xs* 與 *ys* 相對應的元素放在序對中。如果兩個串列長度不同，*zip* 將其中一個用完後即停止。*zip* 也能處理無限長的串列。由於這個動作看來像是把 *xs*與 *ys*當作拉鍊的兩側「拉起來」，因此用拉拉鍊的狀聲詞 “zip” 命名。

相對地，也有一個函數 *unzip* :: List (*a* × *b*) → (List *a* × List *b*)，將「拉鍊」拉開。例：*unzip* [(1,'a'),(2,'b'),(3,'c')] 可得到 ([1,2,3],"abc").

許多情況下，我們不想要把兩兩對應的元素放到序對中，而是分別餵給一個二元運算子。這時可用另一個相關函數 *zipWith*, 例：*zipWith* (+) [1,2,3] [4,5,6] = [5,7,9] 函數 *zipWith* 可以這樣定義：

```
zipWith :: (a → b → c) → List a → List b → List c
zipWith f = map (uncurry f) · zip  .
```

習題 1.14 — 用 *zipWith* 定義 *zip*.

例 1.26. 試定義函數 *positions* :: Char → String → List Int, 使得 *positions z xs* 傳回 *z* 在 *xs* 中出現的所有位置。例: *positions 'o' "hoola hooligans"* = [1,2,7,8].

答. 一種可能寫法如下：

$$positions\ z = map\ fst \cdot filter\ ((== z) \cdot snd) \cdot zip\ [0..]\ .$$

我們用 $zip\ [0..]$ 為輸入串列標上位置，用 $filter\ ((== z) \cdot snd)$ 取出第二個元素等於 z 的序對，最後用 $map\ fst$ 取出所有位置。注意函數合成與 currying 的使用。 □

例 1.27. 接續上例。如果我們僅想要 z 出現的第一個位置呢？我們可以定義：

$$\begin{aligned} &pos :: \mathsf{Char} \to \mathsf{String} \to \mathsf{Int} \\ &pos\ z = head \cdot positions\ z\ . \end{aligned}$$

這是一個部分函數，$pos\ z\ xs$ 傳回 $positions\ z\ xs$ 的第一個結果。如果 z 沒有出現，$positions\ z\ xs$ 傳回 $[\,]$，$pos\ z\ xs$ 會得到執行期錯誤。如果 z 出現在 xs 中，由於惰性求值，pos 得到第一個位置後 $positions$ 便會停下，不會把串列整個產生。

如果我們希望 pos 在 z 沒有出現時傳回 -1，可以這麼做：

$$\begin{aligned} &pos :: \mathsf{Char} \to \mathsf{String} \to \mathsf{Int} \\ &pos\ z\ xs = \mathbf{case}\ positions\ z\ xs\ \mathbf{of} \\ &\quad [\,] \quad\ \to -1 \\ &\quad (i : is) \to i\ . \end{aligned}$$

1.9 全麥編程

讀者至此應已注意到本章採用的特殊編程風格。一般說到串列，大家會先想到資料結構課程中常提到的連結串列 (linked list)。介紹連結串列的範例程式大多用迴圈或遞迴追蹤著指標，一個一個地處理串列中的元素。在指令式語言中做關於陣列的操作時，也常用變數或指標指著「目前的」元素，並在一個迴圈中將該變數逐次遞增或減。總之，我們處理聚合型資料結構時，總是將其中元素一個個取出來處理。但本章的做法不同：我們將整個串列視為一個整體，對整個串列做 *map*, *filter*, *dropWhile* 等動作，或將它和另一個串列整個 *zip* 起來...。

這種編程方式被稱作全麥編程 (*wholemeal programming*)，第 1.11節中將解釋此詞的由來。全麥編程的提倡者們認為：一個個地處理元素太瑣碎，而鼓勵我們拉遠些，使用組件，以更抽象的方式組織程式的結構。

諸如 *map*, *filter*, *iterate*, *zipWith* 等等組件其實都是常見的編程模式。它們可被視為為了特定目的已先寫好的迴圈。拜高階函數與惰性求值之賜，這些組件能容易地被重用在許多不同脈絡中。這麼做的好處之一是：諸如 *map*, *filter*, *zip* 等組件的意義清楚，整個程式的意義也因此會比起自行在迴圈中一個個處理元素來得容易理解。事實上，這麼做可以養成我們思考演算法的新習慣。一些常見的編程模式現在是有名字的，我們把編程模式抽象出來了。而如同第 0章所述，抽象化是我們理解、掌握、操作事物的重要方法。我們現在有了更多詞彙去理解、討論程式與演算法：「這個演算法其實就是先做個 *map*，把結果 *concat* 起來，然後做 *filter*...」

在本書其他章節中我們也將看到：這些抽象化方便我們去操作、轉換程式。具體說來，如果程式用這些組件拼湊成，我們對這些組件知道的性質都可用在我們的程式上。例如，如果我們知道 $map\,f \cdot map\,g = map\,(f \cdot g)$，當我們看到程式中有兩個相鄰的 *map*, 我們可用已知的性質把他們合併成一個 — 這相當於合併兩個迴圈。或著我們可以把一個 *map* 拆成兩個，以方便後續的其他處理。程式的建構方法使得程式含有更多資訊，使我們有更多可操作的空間。

全麥編程之所以成為可能，有賴程式語言的支援。例如，高階函數使得我們能將與特定問題相關的部分（如 $map\,f$ 與 $filter\,p$ 中的 f 與 p）抽象出來；惰性求值使我們勇於使用大串列或無限串列作為中間值，不用擔心它們被不必要地真正算出。

此外，全麥編程也需要豐富的組件函式庫。設計良好的組件捕捉了常見的編程模式，有了它們的幫忙，我們的程式可寫得簡潔明瞭 — 本章之中大部分的程式都是都是一行搞定的 "one-liner". 但，這些組件不可能窮舉所有的編程模式。我們仍會需要自行從頭寫些函數。受到全麥編程影響，在自行寫函數時，我們也常會希望將它們寫得更通用些，藉此發現常見的編程模式，設計出可重用的組件。

全麥編程能寫出多實用的程式？第 1.11節中會提及其他學者嘗試過的，包含解密碼、解數獨在內的有趣例子。在本節，我們則想示範一個小練習：由下至上的合併排序 (merge sort)。

合併排序　假設我們已有一個函數 $merge' :: (\mathsf{List\ Int} \times \mathsf{List\ Int}) \to \mathsf{List\ Int}$, 如果 xs 與 ys 已經排序好，$merge'\,(xs, ys)$ 將它們合併為一個排序好的串列。[14] 函數 $merge'$ 可用第 2.13節的方式歸納寫成，也可使用將在第[todo: where]章提及的組件 *unfoldr* 做出。我們如何用 $merge'$ 將整個串列排序好呢？

一般書中較常提及由上至下的合併排序：將輸入串列（或陣列）切成長度大致相等的兩半，分別排序，然後合併。本節則以由下至上的方式試試看。如果輸入串列為 $[4,2,3,5,8,0,1,7]$，我們先把每個元素都單獨變成串列，也就是變成 $[[4],[2],[3],[5],[8],[0],[1],[7]]$。然後把相鄰的串列兩兩合併：$[[2,4],[3,5],[8,0],[1,7]]$，再兩兩合併成為 $[[2,3,4,5],[0,1,8,7]]$，直到只剩下一個大串列為止。

如果我們定義兩個輔助函數：*wrap* 將一個元素包成一個串列，*isSingle* 判斷一個串列是否只剩下一個元素，

$$\begin{aligned} &wrap :: a \to \mathsf{List}\ a \\ &wrap\ x = [x]\ , \end{aligned} \qquad \begin{aligned} &isSingle :: \mathsf{List}\ a \to \mathsf{Bool} \\ &isSingle\ [x] = \mathsf{True} \\ &isSingle\ xs\ \ = \mathsf{False}\ . \end{aligned}$$

那麼上述的合併排序可以寫成：

$$msort = head \cdot until\ isSingle\ mergeAdj \cdot map\ wrap\ .$$

這幾乎只是把口語描述逐句翻譯：先把每個元素都包成串列，反覆做 *mergeAdj* 直到只剩下一個大串列，然後將那個大串列取出來。

[14] 之所以取名為 $merge'$，因為在第 2.12節中我們將使用一個類似且相關的函數 $merge :: \mathsf{List\ Int} \to \mathsf{List\ Int} \to \mathsf{List\ Int}$.

下一項工作是定義 $mergeAdj :: \text{List (List Int)} \to \text{List (List Int)}$, 其功能是將相鄰的串列兩兩合併。如果我們能訂出一個函數 $adjs :: \text{List } a \to \text{List } (a \times a)$, 將相鄰的元素放在序對中，$mergeAdj$ 就可以寫成：

$$mergeAdj = map\ merge' \cdot adjs \ .$$

但 $adjs$ 該怎麼定義呢？對大部分讀者來說，最自然的方式也許是用第 2章將討論的歸納法。但作為練習，我們姑且用現有的組件試試看。先弄清楚我們對 $adjs$ 的期待。當 $xs = [x_0, x_1, x_2, x_3]$, 我們希望 $adjs\ xs = [(x_0, x_1), (x_2, x_3)]$. 但當 xs 有奇數個元素時，例如 $xs = [x_0, x_1, x_2, x_3, x_4]$, 最後一個元素 x_4 便落單了。如果是為了合併排序，我們也許可以把 x_4 和 $[\,]$ 放在一起，$adjs\ xs = [(x_0, x_1), (x_2, x_3), (x_4, [\,])]$. 但為使 $adjs$ 適用於更多的情況，也許我們應該讓它多拿一個參數，當作落單的元素的配對。因此我們把 $adjs$ 的型別改為 $a \to \text{List } a \to \text{List } (a \times a)$, 希望 $adjs\ z\ xs = [(x_0, x_1), (x_2, x_3), (x_4, z)]$.

我們試著看看這可如何辦到。

- 首先，$zip\ xs\ (tail\ xs)$ 可把 xs 的每個元素和其下一個放在序對中。例：當 $xs = [x_0, x_1, x_2, x_3, x_4]$ 時，$zip\ xs\ (tail\ xs)$ 的值是 $[(x_0, x_1), (x_1, x_2), (x_2, x_3), (x_3, x_4)]$.
- 如果我們為 zip 的第二個參數補上一個 z, 成為 $zip\ xs\ (tail\ (xs +\!\!+ [z]))$，這可歸約為 $[(x_0, x_1), (x_1, x_2), (x_2, x_3), (x_3, x_4), (x_4, z)]$。
- 再將位置（由 0 算起）為奇數的元素丟棄，我們便得到原先希望的 $[(x_0, x_1), (x_2, x_3), (x_4, z)]$ 了！

讀者可試試看當 xs 有偶數個元素時的情況。總之，$adjs$ 可定義成：

$$\begin{aligned} &adjs :: a \to \text{List } a \to \text{List } (a \times a) \\ &adjs\ z\ xs = everyother\ (zip\ xs\ (tail\ xs +\!\!+ [z])) \ , \end{aligned}$$

其中 $everyother\ ys$ 把 ys 中位置為奇數的元素丟棄。

最後，考慮如何把串列中位置為奇數的元素丟棄。一種做法是：一直從串列中丟掉頭兩個元素，直到串列用完：

$$\begin{aligned} &everyother :: \text{List } a \to \text{List } a \\ &everyother = map\ head \cdot takeWhile\ (not \cdot null) \cdot iterate\ (drop\ 2) \ . \end{aligned}$$

總而言之，由下至上的合併排序可寫成：

$$\begin{aligned} &msort :: \text{List Int} \to \text{List Int} \\ &msort = head \cdot until\ isSingle\ mergeAdj \cdot map\ wrap \ , \end{aligned}$$

其中 $mergeAdj$ 的定義是：

$$\begin{aligned} &mergeAdj :: \text{List (List Int)} \to \text{List (List Int)} \\ &mergeAdj = map\ merge' \cdot adjs\ [\,] \ . \end{aligned}$$

如果我們想看到合併排序完成前的每一步驟，可將 $msort$ 中（以 $iterate$ 與 $dropWhile$ 定義出）的 $until$ 改為 $iterate$ 與 $takeWhile$:

```
msortSteps :: List Int → List (List (List Int))
msortSteps = takeWhile (not · isSingle) · iterate mergeAdj · map wrap .
```

例如，*msortSteps* [9,2,5,3,6,4,7,0,5,1,8,2,3,1] 可得到

```
[[[9],[2],[5],[3],[6],[4],[7],[0],[5],[1],[8],[2],[3],[1]],
 [[2,9],[3,5],[4,6],[0,7],[1,5],[2,8],[1,3]],
 [[2,3,5,9],[0,4,6,7],[1,2,5,8],[1,3]],
 [[0,2,3,4,5,6,7,9],[1,1,2,3,5,8]]] .
```

最後兩個串列合併為 [0,1,1,2,2,3,3,4,5,5,6,7,8,9]，即為 *msort* 的結果。

1.10 自訂資料型別

本章目前為止給讀者看到的 **data** 定義其實都是 Haskell 已內建的型別。使用者也可自己定義新資料型別。例如，我們可能定義一個新型別表達四個方向：

```
data Direction = North | East | South | West ,
```

或著定義一個表示顏色的型別，用三個浮點數表達紅、綠、藍的比例：

```
data RGBColor = RGB Float Float Float ,
```

例：土耳其藍 (turquoise) 可寫成 RGB 0.25 0.875 0.8125. 下列函數則降低一個顏色的彩度：[15]

```
desaturate :: Float → RGBColor → RGBColor
desaturate p (RGB r g b) =
  RGB (r+p×(gr−r)) (g+p×(gr−g)) (b+p×(gr−b)) ,
  where gr = r×0.299+g×0.587+b×0.144 .
```

我們也可定義如 List 一樣的遞迴資料型別。例如，資料結構中可能談到兩種二元樹狀結構，一種僅在內部節點有標示（稱作 internally labelled），另一種僅在葉節點有表示（稱作 externally labelled）。這兩種二元樹可分別表示如下：

```
data ITree a  = Null | Node a (ITree a) (ITree a) ,
data ETree a = Tip a | Bin (ETree a) (ETree a) .
```

怎麼編寫這種資料結構上的程式呢？我們將在下一章中說到。

1.11 參考資料

本章中的許多想法取自 Bird [1998] ，該書是我相當推薦的 Haskell 教材。

[15] 這是一個簡便的做法：算出該顏色的灰度 *gr*, 然後計算每個原色與該灰度的線性內插。更準確的作法應將 RGB 轉成 HSV，以後者調整飽和度。

Haskell 為何叫 Haskell?

1980 年代中期，程式語言學者們已各自開發出了許多個語法、語意類似但卻稍有不同、大都只在出生機構被使用的惰性純函數語言。沒有一個語言取得壓倒性的優勢。為溝通方便、以及為了讓整個領域能走向下一步，大家有了該設計個統合、共通的惰性純函數語言的共識。1988 年一月，新語言設計小組在耶魯大學開會，眾多討論項目中包括幫語言取個名字。以下軼事節錄自 Hudak et al. [2007] .

當天被提出的選項包括 Haskell, Vivaldi, Mozart, CFL (Common Functional Language), Curry, Frege, Peano, Nice, Fun, Light... 等等。最後經程序選出的名字是 "Curry", 紀念邏輯學家 Haskell B. Curry — 他在組件邏輯 (combinatorial logic)、Curry-Howard 同構等領域的研究一直深遠影響函數語言學界。

但當天晚上就有人覺得這名字會招惹太多雙關語笑話。除了咖哩之外，小組成員覺得實在不行的是：TIM (three instruction machine) 是函數語言用的一種抽象機器，但 Tim Curry 則成了電影洛基恐怖秀（Rocky Horror Picture Show, 1975）的男主角。

於是新語言的名字就改成 Haskell 了。

小組成員 Paul Hudak 和 David Wise 寫信給 Curry 的遺孀 Virginia Curry，徵求她的同意。Hudak 後來親自登門拜訪，Virginia Curry 和他聊了之前的訪客（包含 Church 與 Kleene）的故事；後來她也去聽了 Hudak 關於 Haskell （語言）的演講，表現得十分友善。臨別前，她說：「其實呀，Haskell 一直都不喜歡他的名字。」

Currying Moses [Schönfinkel, 1924] 提出多參數函數可用單參數函數表達。Haskell Curry 在許多著作中（例：Curry [1980] ）使用 currying，但當時並沒有 currying 一詞。為何此概念最後會以 Curry 命名呢？David A. Turner （Haskell 語言的前身之一 Miranda 的設計人）在一次網路討論 [Sankar et al., 1997] 中表示 currying 一詞由 Christopher Strachey 取名，於 1967 年前後使用在其上課資料中。這種說法目前廣被大家接受，但我目前尚未找可佐證的上課資料。相反地，Strachey [1967] 之中明確表示他認為 currying 的概念是由 Schönfinkel 發明的，並稱之為「Schönfinkel 的裝置」。[16] 但 currying 的想法可追溯得比 Schönfinkel 或 Curry 都早。F. L. Gottlob Frege 1891 年的 Über Funktion und Begriff (英譯 Function and Concept) [Frege, 1960] 結尾幾頁的概念即是 currying.

全麥編程 「全麥編程」一詞由牛津大學 Geraint Jones 取名，由來可能是模仿健康食物的說詞。如 Bird [2010, 第 19 章] 便寫道，「全麥編程好處多多，可預防『索引症』(indexitis)，鼓勵合法的程式建構。」在該章之中，Richard Bird 以大量使用串列組件函數的全麥編程為起點，推導出能相當迅速地解數獨的程式。Hinze [2009] 以全麥編程為工具，示範了河內塔問題 (Tower of Hanoi) 的許多性質，以及其與謝爾賓斯基 (Sierpiński) 三角形的關係。其中寫道「函數語言擅長全麥編程。這個詞彙由 Geraint Jones 命名。全麥編程意謂由大處去想：處理整個串列，而不是一連串的元素；發展出整個解答的空間，而不是個別的解答；想像整個圖，而不是單一的路徑。對於已知的問題，全麥編程常給我們新洞察與新觀點。」Hutton [2016, 第五章] 則以編、解密碼為例。凱撒加密 (*Caesar cipher*) 為一種簡單的加密方式：將明文中的每個字母都往前或後偏移固定的

[16] 原文："There is a device originated by Schönfinkel, for reducing operators with several operands to the successive application of single operand operators."

量，例如當偏移量為 2 時，`'a'` 變成 `'c'`，`'b'` 變成 `'d'` ... 一種解凱撒密碼的有效方式是計算密文中每個字母的分佈，和一般英文文章中的平均字母分佈做比較，藉以猜出偏移量。Graham Hutton 在書中示範如何用組件函數、完全不用遞迴地寫出編碼與解碼程式。以上都是相當值得一看的例子。

LISP 的串列 誕生於 1958 年的 LISP 是目前仍被使用的高階程式語言中歷史第二悠久的 — 最早的是 FORTRAN. 但 LISP 與 FORTRAN 是風格截然不同的語言。雖然具有含副作用的指令，LISP 仍被認為是函數語言的先驅。

LISP 為「串列處理 (list processing)」的縮寫。但事實上，LISP 中的聚合資料結構「S 算式 (S-expression)」不只可用來表達串列。CONS 函數做出的是一個序對，其中第一個元素被稱作 CAR (contents of the address part of register), 第二個稱作 CDR (contents of the decrement part of register). 如果 CDR 的部分仍是一個 CONS 做出的序對，或是特殊值 NIL, 整個結構表達的就是一個串列。S 算式也可用來做出二元樹、語法樹... 等等。Haskell 串列的建構元 `[]` 可唸成 'nil", `(:)` 唸成 'cons", 這兩個詞彙都從 LISP 而來。

在前幾波人工智慧熱潮時，大家認為符號與邏輯的處理是人工智慧的基礎。但早期的程式語言大多針對數值運算而設計，會處理串列的 LISP 便被視為最適合做符號處理的語言 —「人工智慧用的語言」。另一個被視為「人工智慧專用語言」的是奠基於述語邏輯與歸結 (resolution) 的 PROLOG. 今日的人工智慧技術以神經網路為基礎，「人工智慧專用語言」的頭銜則給了 Python。

CHAPTER 2

歸納定義與證明

「全麥編程」的觀念鼓勵我們以小組件組織出大程式。但這些個別組件該如何實作呢？或著，沒有合用的組件時該怎麼辦？我們可以回到更基礎的層次，用遞迴 (recursion)定義它們。「遞迴」意指一個值的定義又用到它本身，是數學中常見的定義方式。在早期的編程教材中，遞迴常被視為艱澀、難懂、進階的主題。但在函數程設中，遞迴是唯一可使程式不定次數地重複一項計算的方法。一旦跨過了門檻，遞迴其實是個很清晰、簡潔地描述事情的方式。

對於遞迴，許多初學者一方面不習慣、覺得如此構思程式很違反「直覺」，另一方面也納悶：以自己定義自己，到底是什麼意思？這兩個難處其實都談到了好問題。對於前者，我們希望發展一些引導我們構思遞迴程式的思路；希望有了這些依據，能使寫遞迴程式變得直覺而自然。關於後者，其實並非所有遞迴定義都有「意思」— 有些「不好」的遞迴並沒有定義出東西。我們討論遞迴的意義時必須限定在「好」的、有意義的程式上。最好有些方式確保我們寫出的遞迴定義是好的。

在本章我們將討論一種型式較單純的遞迴：歸納 (induction)。對上述兩個問題，本章的回應是：先有歸納定義出的資料結構，再依附著該資料結構撰寫歸納定義的程式，是一種思考、解決程式問題的好方法，也是一種理解遞迴程式的方式。此外，依循這種方法也能確保該定義是「好」的。我們將從數學歸納法出發，發現歸納程式與數學歸納法的相似性 — 寫程式和證明其實是很相似的活動。

本書以 Haskell 為學習工具，但在之後的幾章，我們僅使用 Haskell 的一小部分。Haskell 支援無限大的資料結構，也允許我們寫出不會終止的程式。但我們將假設所有資料結構都是有限的（除非特別指明），所有函數皆會好好終止（也就是說函數都是「全函數 (total function)」— 對每一個值，都會好好地算出一個結果，不會永遠算下去，也不會丟回一個錯誤）。這麼做的理由將在本章解釋。

2.1 數學歸納法

在討論怎麼寫程式之前，我們得先複習一下*數學歸納法* — 晚點我們就會明白理由。回顧：*自然數*在此指的是 $0, 1, 2...$ 等所有非負整數。[1] 自然數有無限多個，但每個自然數都是有限大的。自然數的型別記為 $\mathbb{N}$. 若 a是一個型別, a之上的*述語* (predicate)可想成型別為 $a \to \mathsf{Bool}$的函數，常用來表示某性質對某特定的 a是否成立。自然數上的述語便是 $\mathbb{N} \to \mathsf{Bool}$。數學歸納法可用來證明某性質對所有自然數都成立：

> 給定述語 $\mathsf{P} :: \mathbb{N} \to \mathsf{Bool}$. 若
>
> 1. P 對 0 成立，並且
> 2. 若 P 對 n 成立，P 對 $1 + n$ 亦成立，
>
> 我們可得知 P 對所有自然數皆成立。

為何上述的論證是對的？我們在 2.9節將提供一個解釋。但此處我們可以提供一個和程式設計較接近的理解方式。自然數可被想成如下的一個資料結構：

$$\mathbf{data}\ \mathbb{N} = \mathbf{0} \mid \mathbf{1}_+\ \mathbb{N}\ .$$

這行定義有幾種讀解法，目前我們考慮其中一種 — 該定義告訴我們：

1. $\mathbf{0}$ 的型別是 $\mathbb{N}$;
2. 如果 n 的型別是 $\mathbb{N}$, $\mathbf{1}_+\ n$ 的型別也是 $\mathbb{N}$;
3. 此外，沒有其他型別是 $\mathbb{N}$ 的東西。

這種定義方式稱作*歸納定義* (inductive definition)。其中「沒有其他型別是 $\mathbb{N}$ 的東西」一句話很重要 — 這意味著任一個自然數只可能是 $\mathbf{0}$，或是另一個自然數加一，沒有別的可能。任一個自然數都是這麼做出來的：由 $\mathbf{0}$ 開始，套上*有限*個 $\mathbf{1}_+$。反過來說，給任意一個自然數，我們將包覆其外的 $\mathbf{1}_+$ 一層層拆掉，在有限時間內一定會碰到 $\mathbf{0}$. 有人主張將 inductive definition 翻譯為*迭構定義*，著重在從基底（此處為 $\mathbf{0}$）開始，一層層堆積上去（此處為套上 $\mathbf{1}_+$）的概念。

本書中，我們把自然數的 $\mathbf{0}$ 寫成粗體，表明它是資料建構元；把 $\mathbf{1}_+$ 的加號寫得小些並和 $\mathbf{1}$ 放得很近，以強調「加一」是資料建構元、是一個不可分割的動作（和我們之後將介紹的一般自然數加法 $(+)$不同）。數字 2 其實是 $\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})$ 的簡寫, 3 其實是 $\mathbf{1}_+\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0}))$ 的簡寫。

歸納定義出的資料型別允許我們做*歸納證明*。由於 P 是 $\mathbb{N}$ 到 Bool 的函數，「P 對 0 成立」可記為 $\mathsf{P}\ \mathbf{0}$，「若 P 對 n 成立，P 對 $1 + n$ 亦成立」可記為 $\mathsf{P}\ (\mathbf{1}_+\ n) \Leftarrow \mathsf{P}\ n$。[2] 我們假設兩者都已被證明，用它們證明 $\mathsf{P}\ 3$：

$$\begin{array}{ll} & \mathsf{P}\ (\mathbf{1}_+\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0}))) \\ \Leftarrow & \quad \{\, 因\ \mathsf{P}\ (\mathbf{1}_+\ n) \Leftarrow \mathsf{P}\ n \,\} \\ & \mathsf{P}\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})) \\ \Leftarrow & \quad \{\, 因\ \mathsf{P}\ (\mathbf{1}_+\ n) \Leftarrow \mathsf{P}\ n \,\} \\ & \mathsf{P}\ (\mathbf{1}_+\ \mathbf{0}) \end{array}$$

[1] 有些數學派別的「自然數」是從 1 算起的。計算科學中則通常習慣以 0 起算。

[2] $\mathsf{P} \Leftarrow \mathsf{Q}$ 意思是「若 Q 則 P」。依此順序寫，有「為證明 P，我們想辦法讓 Q 成立」的感覺。許多人不習慣由右到左的箭頭，但不論數學上或日常生活中，這都是常使用的論證思考方向。

$$\begin{array}{ll} \Leftarrow & \{\text{因 } \mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n\} \\ \quad \mathsf{P}\,\mathbf{0}\ . \end{array}$$

第一步中，我們希望 $\mathsf{P}\,(\mathbf{1}_+\,(\mathbf{1}_+\,(\mathbf{1}_+\,\mathbf{0})))$ 成立，根據 $\mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n$, 只要 $\mathsf{P}\,(\mathbf{1}_+\,(\mathbf{1}_+\,\mathbf{0}))$ 即可。第二步中，我們希望 $\mathsf{P}\,(\mathbf{1}_+\,(\mathbf{1}_+\,\mathbf{0}))$ 成立，同樣根據 $\mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n$，只要 $\mathsf{P}\,(\mathbf{1}_+\,\mathbf{0})$ 成立即可... 最後，只要 $\mathsf{P}\,\mathbf{0}$ 成立，$\mathsf{P}\,(\mathbf{1}_+\,\mathbf{0})$ 即成立，但 $\mathsf{P}\,\mathbf{0}$ 是已知的。因此我們已論證出 $\mathsf{P}\,3$ 成立！

由上述推演中，我們發現：數學歸納法的兩個前提 $\mathsf{P}\,\mathbf{0}$ 與 $\mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n$ 給了我們一個對任一個自然數 m, 生成一個 $\mathsf{P}\,m$ 之證明的方法。這是由於自然數本就是一個歸納定義出的資料型別：任一個自然數 m 都是有限個 $\mathbf{1}_+$ 套在 $\mathbf{0}$ 之上的結果，因此，只要反覆用 $\mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n$ 拆，總有碰到 $\mathsf{P}\,\mathbf{0}$ 的一天。既然對任何 m, 都做得出一個 $\mathsf{P}\,m$ 的證明，我們就可安心相信 P 對任何自然數都成立了。

為了之後討論方便，我們將前述的數學歸納法寫得更形式化些：

$$\text{自然數上之歸納法：}\ (\forall n \cdot \mathsf{P}\,n) \ \Leftarrow\ \mathsf{P}\,\mathbf{0} \wedge (\forall n \cdot \mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n)\ .$$

這只是把之前的文字描述改寫成二階邏輯，但可清楚看出：給定 P, 我們希望證明它對所有自然數都成立，只需要提供 $\mathsf{P}\,\mathbf{0}$ 和 $\mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n$ 兩個證明。其中 $\mathsf{P}\,\mathbf{0}$ 是確定 P 對 0 成立的基底 (base case)，$\mathsf{P}\,(\mathbf{1}_+\,n) \Leftarrow \mathsf{P}\,n$ 則被稱作歸納步驟 (inductive step)：在假設 $\mathsf{P}\,n$ 成立的前提下，想辦法「多做一步」，論證 $\mathsf{P}\,(\mathbf{1}_+\,n)$ 也成立。餘下的就可交給數學歸納法這個架構了。

2.2 自然數上之歸納定義

數學歸納法和編程有什麼關係呢？考慮一個例子：給定 $b, n :: \mathbb{N}$, 我們希望寫個函數 exp 計算乘冪，使得 $exp\ b\ n = b^n$. 我們先把型別寫下：

$$\begin{array}{l} exp :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ exp\ b\ n = ? \end{array}$$

問號部分該怎麼寫？沒有其他線索很難進行，因此我們回想：n 是自然數，而任何自然數只可能是 $\mathbf{0}$ 或 $\mathbf{1}_+$ 做出的。我們便分成這兩個狀況考慮吧：

$$\begin{array}{l} exp :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ exp\ b\ \mathbf{0} \qquad = ? \\ exp\ b\ (\mathbf{1}_+\,n) = ? \end{array}$$

其中，$exp\ b\ \mathbf{0}$ 較簡單：顯然應該是 $b^0 = 1$. 至於 $exp\ b\ (\mathbf{1}_+\,n)$ 的右手邊該怎麼寫？似乎很難一步定出來。但假設 $exp\ b\ n$ 已經順利算出了 b^n, 由於 $b^{1+n} = b \times b^n$, $exp\ b\ (\mathbf{1}_+\,n)$ 與之的關係可寫成：

$$exp\ b\ (\mathbf{1}_+\,n) = b \times exp\ b\ n\ .$$

如此一來我們便完成了一個計算乘冪的程式：

Haskell v.s Math

很不幸地，Haskell 並不接受 2.2節中 exp的定義。

首先，Haskell 並沒有獨立的自然數型別。我們可自己定（並將其宣告為 Num 類別的一員），或著直接使用 Haskell 內建的 Int 型別。其次，Haskell 原本允許我們在定義的左手邊寫 $exp\ b\ (n+1)$ ，但這套稱作 "$n+k$ pattern" 的語法已在 Haskell 2010 中被移除。目前我們得將 exp 寫成：

$$\begin{aligned} &exp :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ &exp\ b\ 0 = 1 \\ &exp\ b\ n = b \times exp\ b\ (n-1)\ . \end{aligned}$$

$n+k$ pattern 曾引起激烈討論。支持者主要著眼於它在教學上的方便：這方便我們討論數學歸納法、做證明、並讓我們更明顯地看出自然數與串列的相似性。反對者則批評它與 type class 的衝突。後來由反方勝出。

有些 Haskell 教科書堅持書中出現的程式碼須是能在一個字一個字地鍵入電腦後即可執行的。本書的定位並非 Haskell 教材，而是函數編程概念的入門書。為此目的，我們希望選擇適合清晰表達概念、易於操作、演算、證明的符號。而一個實用目的的語言得在許多設計上妥協尋求平衡，基於種種考量，往往得犧牲符號的簡潔與便利性（這點我們完全能理解）。因此本書中的程式語法偶爾會和 Haskell 語法有所不同。我們會盡量指明這些不同處，使讀者知道如何將本書中的程式轉換成現下的 Haskell 語法。

$$\begin{aligned} &exp :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &exp\ b\ \mathbf{0} \qquad = 1 \\ &exp\ b\ (\mathbf{1}_+\ n) = b \times exp\ b\ n\ . \end{aligned}$$

回顧一下剛剛的思路：我們難以一步登天地對任何 n 寫出 $exp\ b\ n$, 但我們提供 $exp\ b\ \mathbf{0}$ 該有的值，並在假設 $exp\ b\ n$ 已算出該有的值的前提下，試著做一點加工、多算那一步，想法做出 $exp\ b\ (\mathbf{1}_+\ n)$ 該有的值。這和前述的數學歸納法是一樣的！寫歸納程式和做歸納證明是很類似的行為。使用數學歸納法證明 P 需要提供一個基底 P 0 和歸納步驟 P $(\mathbf{1}_+\ n) \Leftarrow$ P n. 歸納定義程式也一樣。在 $exp\ b\ n$ 的定義中，基底是 $exp\ b\ \mathbf{0}$，歸納步驟則是由 $exp\ b\ n$ 想法變出 $exp\ b\ (\mathbf{1}_+\ n)$. 有這兩個元件，我們便有了一個對任何自然數 n, 保證算出 $exp\ b\ n$ 的方法。作為例子，我們看看 $exp\ 2\ 3$ 是怎麼被算出來的：

$$\begin{aligned} & exp\ 2\ (\mathbf{1}_+\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0}))) \\ = & \quad \{ exp \text{ 之歸納步驟} \} \\ & 2 \times exp\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})) \\ = & \quad \{ exp \text{ 之歸納步驟} \} \\ & 2 \times 2 \times exp\ (\mathbf{1}_+\ \mathbf{0}) \\ = & \quad \{ exp \text{ 之歸納步驟} \} \\ & 2 \times 2 \times 2 \times exp\ \mathbf{0} \\ = & \quad \{ exp \text{ 之基底} \} \\ & 2 \times 2 \times 2 \times 1\ . \end{aligned}$$

第一步中，要算出 $exp\ 2\ (\mathbf{1}_+\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})))$, 我們得先算出 $exp\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0}))$. 要算出後者，在第二步中我們得先算出 $exp\ (\mathbf{1}_+\ \mathbf{0})$... 直到我們碰到 $exp\ b\ \mathbf{0}$.

自然數上的歸納定義 我們將 b 固定，稍微抽象一點地看 $exp\ b :: \mathbb{N} \to \mathbb{N}$ 這個函數。該定義符合這樣的模式：

$$
\begin{aligned}
&f :: \mathbb{N} \to a \\
&f\ \mathbf{0} \quad\quad = e \\
&f\ (\mathbf{1}_+\ n) = \ldots f\ n \ldots \quad .
\end{aligned}
$$

這類函數的輸入是 $\mathbb{N}$，其定義中 $f\ (\mathbf{1}_+\ n)$ 的狀況以 $f\ n$ 定出，此外沒有其他對 f 的呼叫。若一個函數符合這樣的模式，我們說它是*在自然數上歸納定義*出的，其中 $f\ \mathbf{0}$ 那條稱作其基底，$f\ (\mathbf{1}_+\ n)$ 那條稱作其歸納步驟。我們日後將看到的許多程式都符合這個模式。

數學上，若一個函數能為其值域內的每個值都找到一個輸出，我們說它是個全函數 (total function)，否則是部分函數 (partial function)。計算上，當我們說 f 是一個全函數，意謂只要 x 型別正確並可算出值，$f\ x$ 便能終止並算出一個值，不會永久跑下去，也不會丟出錯誤。

如前所述的、在自然數上歸納定義的 f 會是全函數嗎？首先，任何自然數都可拆成 $\mathbf{0}$ 或是 $\mathbf{1}_+\ n$，而這兩個情況已被 f 的兩行定義涵括，不會出現漏失的錯誤。其次，f 每次呼叫自己，其參數都少了一層 $\mathbf{1}_+$。長此以往，不論輸入多大，總有一天會遇到基底 $f\ \mathbf{0}$ — 因為任何自然數都是從 $\mathbf{0}$ 開始，套上有限個 $\mathbf{1}_+$。只要基底狀況的 e 以及在歸納步驟中 $f\ n$ 前後的計算都正常終止，對任何輸入，f 都會正常終止。因此 f 是個全函數。

「程式會終止」是很重要的性質，我們之後會常談到。在本書目前為止示範的編程方法中，「一個函數若呼叫自己，只能給它更小的參數」是個單純但重要的規範（例如 $f\ (\mathbf{1}_+\ n)$ 的右手邊可以有 $f\ n$，不能有 $f\ (\mathbf{1}_+\ n)$ 或 $f\ (\mathbf{1}_+\ (\mathbf{1}_+\ n))$）。操作上這確保程式會終止，而在第 [todo: where?] 章之中，我們將提到這也確保該遞迴定義是「好」的、有意義的。

順便一提：在 $f\ (\mathbf{1}_+\ n)$ 的右手邊中，$f\ n$ 可以出現不只一次 — 因為 $\ldots f\ n \ldots f\ n \ldots$ 可看成 $(\lambda x \to \ldots x \ldots x \ldots)\ (f\ n)$。在 $f\ n$ 的前後 $\ldots$ 的部分可以出現 n — 將在第60頁中介紹的階層函數就是一個這樣的例子。有些情況下我們不允許 n 出現在 $\ldots$ 中，此時會額外說明。

乘法、加法 我們多看一些歸納定義的例子。在 exp 中我們用到乘法，但假若我們的程式語言中只有加法、沒有乘法呢？我們可自己定定看：

$$
\begin{aligned}
&(\times) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&m \times n = ?
\end{aligned}
$$

若不用組件，我們目前會的寫程式方法只有歸納法，也只有這招可試試看了。但，$(\times)$ 有兩個參數，我們該把 $(m\times) :: \mathbb{N} \to \mathbb{N}$ 視為一個函數，分別考慮 n 是 $\mathbf{0}$ 或 $\mathbf{1}_+\ldots$ 的情況，還是把 $(\times n) :: \mathbb{N} \to \mathbb{N}$ 視為一個函數，考慮 m 是 $\mathbf{0}$ 或 $\mathbf{1}_+\ldots$ 的情況？答案是兩者皆可，並無根本性的差異。只是現在我們做的選擇會影響到之後與 $(\times)$ 相關的證明怎麼寫 (見第 2.3節)。本書中的習慣是拆左手邊的參數，因此我們考慮以下兩種情況。

$$
\begin{aligned}
&(\times) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathbf{0} \qquad \times n = ? \\
&(\mathbf{1}_+ \, m) \times n = \ldots m \times n \ldots .
\end{aligned}
$$

基底狀況中，$\mathbf{0} \times n$ 的合理結果應是 $\mathbf{0}$. 歸納步驟中，我們得想法算出 $(\mathbf{1}_+ \, m) \times n$, 但我們可假設 $m \times n$ 已經算出了。稍作思考後，讀者應可同意以下的做法：

$$
\begin{aligned}
&(\times) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathbf{0} \qquad \times n = \mathbf{0} \\
&(\mathbf{1}_+ \, m) \times n = n + (m \times n) \ ,
\end{aligned}
$$

如果已有 $m \times n$，多做一個 $(n+)$, 就可得到 $(\mathbf{1}_+ \, m) \times n$了。

如果我們的程式語言中連加法都沒有呢？加法可看成連續地做 $\mathbf{1}_+$:

$$
\begin{aligned}
&(+) :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathbf{0} \qquad + n = n \\
&(\mathbf{1}_+ \, m) + n = \mathbf{1}_+ \, (m + n) \ .
\end{aligned}
$$

此處 $(+)$ 是我們定義的、可將任意兩個自然數相加的加法，而 $\mathbf{1}_+$ 只做「加一」，是基本的資料建構元。為求一致，我們同樣在左邊的參數上做歸納。基底狀況中，$\mathbf{0} + n$ 只應是 n. 想計算 $(\mathbf{1}_+ \, m) + n$, 先假設 $m + n$ 已經算出，再多套一個 $\mathbf{1}_+$. 不難看出 $m + n$ 是把 n 當做基底，在外面套上 m 個 $\mathbf{1}_+$ 的結果。[3]

2.3 自然數上之歸納證明

上一節中我們定出了函數 exp。如果定義正確，$exp \; b \; n$ 算的應是 b^n. 例如，我們知道 $b^{m+n} = b^m \times b^n$. 我們定出的函數 exp 是否真有此性質呢？

定理 2.1. 對任何 $b, m, n :: \mathbb{N}$, $exp \; b \; (m + n) = exp \; b \; m \times exp \; b \; n$.

我們試著證明定理 2.1。數學歸納法是我們目前唯一的工具，而要使用它，第一個問題是：該用 b, m, 或 n 的哪一個來做歸納呢（意即把哪一個拆解）？

觀察定理 2.1中待證明式的等號兩邊，並參照 exp, $(+)$, 與 $(\times)$的定義。等號左手邊的 $exp \; b \; (m + n)$ 之中，化簡 $exp \; b$ 前得知道 $m + n$ 究竟是 $\mathbf{0}$ 還是 $\mathbf{1}_+ \, k$。而根據 $(+)$ 的定義，化簡 $m + n$ 前需知道 m 究竟是 $\mathbf{0}$ 還是 $\mathbf{1}_+ \, k$. 再看右手邊，根據 $(\times)$ 的定義，要化簡 $exp \; b \; m \times exp \; b \; n$ 得先化簡 $exp \; b \; m$, 而後者也得知道 m 是什麼。對兩邊的分析都指向：我們應針對 m 做歸納！

策略擬定後，我們便試試看吧！

證明定理 *thm:exp-plus-times.* 欲證明 $exp \; b \; (m + n) = exp \; b \; m \times exp \; b \; n$, 我們在 m 之上做歸納。m 要不就是 $\mathbf{0}$, 要不就是 $\mathbf{1}_+ \, k$.

情況 $m := \mathbf{0}$. 此時需證明 $exp \; b \; (\mathbf{0} + n) = exp \; b \; \mathbf{0} \times exp \; b \; n$. 推論如下：

$$
\begin{aligned}
& exp \; b \; (\mathbf{0} + n) \\
= \quad & \{ (+) \text{ 之定義} \}
\end{aligned}
$$

[3] 「這樣做不是很慢嗎？」是的。本章的自然數表示法，以及其引申出的運算元都不應看作有效率的實作，而是理論工具。了解加法與乘法可這樣看待後，許多其相關性質都可依此推導出來。

$$
\begin{aligned}
& exp\ b\ n \\
= & \quad \{\text{因 } 1 \times k = k\} \\
& 1 \times exp\ b\ n \\
= & \quad \{exp \text{ 之定義}\} \\
& exp\ b\ \mathbf{0} \times exp\ b\ n \ .
\end{aligned}
$$

情況 $m := \mathbf{1}_+\ m$. 此時需證明 $exp\ b\ ((\mathbf{1}_+\ m) + n) = exp\ b\ (\mathbf{1}_+\ m) \times exp\ b\ n$, 但可假設 $exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n$ 已成立。推論如下：

$$
\begin{aligned}
& exp\ b\ ((\mathbf{1}_+\ m) + n) \\
= & \quad \{(+) \text{ 之定義}\} \\
& exp\ b\ (\mathbf{1}_+\ (m+n)) \\
= & \quad \{exp \text{ 之定義}\} \\
& b \times exp\ b\ (m+n) \\
= & \quad \{\text{歸納假設}\} \\
& b \times (exp\ b\ m \times exp\ b\ n) \\
= & \quad \{(\times) \text{ 之結合律}\} \\
& (b \times exp\ b\ m) \times exp\ b\ n \\
= & \quad \{exp \text{ 之定義}\} \\
& exp\ b\ (\mathbf{1}_+\ m) \times exp\ b\ n \ .
\end{aligned}
$$

□

對這個證明，讀者是否有所懷疑？最大的疑問可能在「假設 $exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n$ 成立」這句上。這不就是我們要證明的性質嗎？在證明中假設它成立，似乎是用該性質自己在證明自己。這是可以的嗎？

為清楚說明，我們回顧一下第 2.1節中的數學歸納法（並把區域識別字改為 k 以避免混淆）：

$$\text{自然數上之歸納法：}\ (\forall k \cdot \mathsf{P}\ k) \Leftarrow \mathsf{P}\ \mathbf{0} \wedge (\forall k \cdot \mathsf{P}\ (\mathbf{1}_+\ k) \Leftarrow \mathsf{P}\ k) \ .$$

證明 2.1欲證的是 $exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n$，並在 m 上做歸納。更精確地說，就是選用了下述的 P:[4]

$$\mathsf{P}\ m \equiv (exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n) \ ,$$

在證明中改變的是 m，而 b 與 n 是固定的。數學歸納法可證明 $(\forall m \cdot \mathsf{P}\ m)$, 展開後正是 $(\forall m \cdot exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n)$. 而根據數學歸納法，我們需提供 $\mathsf{P}\ \mathbf{0}$ 與 $(\forall m \cdot \mathsf{P}\ (\mathbf{1}_+\ m) \Leftarrow \mathsf{P}\ m)$ 的證明。

證明 2.1中「**情況** $m := \mathbf{0}$」的部分，就是 $\mathsf{P}\ \mathbf{0}$ 的證明。而「**情況** $m := \mathbf{1}_+\ m$」則是 $(\forall m \cdot \mathsf{P}\ (\mathbf{1}_+\ m) \Leftarrow \mathsf{P}\ m)$ 的證明。「假設 $exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n$ 成立」指的是假設 $\mathsf{P}\ m$ 成立，我們在此前提之下試圖證明 $\mathsf{P}\ (\mathbf{1}_+\ m)$. 因此，證明 2.1 並沒有「用該性質自己證明自己」。我們是以一個比較小的結果（$\mathsf{P}\ m$）證明稍大

[4]在程式推導圈子中，$(\equiv)$ 常用來代表「只用在真假值上、且滿足結合律的等號」。本書中使用 $(\equiv)$ 以和 $(=)$ 做區分。

一點的結果（$P\ (\mathbf{1}_+\ m)$）。就如同我們寫歸納程式時，假設 $f\ n$ 已經算出，試著用它定出 $f\ (\mathbf{1}_+\ n)$.

在證明之中，如 $P\ m$ 這種在歸納步驟假設成立的性質被稱作歸納假設 (induction hypothesis)。

程式與證明 證明 2.1還有一些能啟發我們之處。方才，我們看到 $exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n$，決定以數學歸納法證明，但接下來怎麼著手？怎麼選定在哪個變數上做歸納？

答案是：分析該式中程式的行為。程式怎麼拆其參數，我們在證明中就怎麼拆。我們試圖證明某些程式的性質，但程式本身便提供了證明可怎麼進行的提示。「使證明的結構符合程式的結構」是許多證明的秘訣。並非所有證明都可以如此完成，但本原則在許多情況下適用。

再看看 $exp, (+), (\times)$ 等函數的定義。為何他們都分出了兩個情況：$\mathbf{0}$ 與 $\mathbf{1}_+\ n$，並且 $\mathbf{1}_+\ n$ 的情況使用到該函數對於 n 的值？因為自然數的資料型別是這麼定的！自然數只可能是 $\mathbf{0}$ 或 $\mathbf{1}_+\ n$，而後者是由 n 做出來的。因此程式也如此寫。程式的結構依循與其處理的資料型別之結構。

資料、程式、與證明原來有著這樣的關係：*證明的結構依循著程式的結構，而程式的結構又依循著資料型別的結構*。歸納定義出了一個型別後，自然知道怎麼在上面寫歸納程式；歸納程式有了，自然知道如何做關於這些程式的歸納證明。一切由定義資料結構開始。掌握這個原則，大部分的證明就不是難事。

讓符號為你工作 再考慮證明 2.1中的狀況 $m := \mathbf{1}_+\ m$. 假想由你做這個證明，由第一行 $exp\ b\ ((\mathbf{1}_+\ m)+n)$ 開始。接下來該怎麼進行？

既然我們已經打定主意用數學歸納法，在證明的某處必定會使用 $exp\ b\ (m+n) = exp\ b\ m \times exp\ b\ n$這個歸納假設。因此，證明前幾行的目的便是想辦法將 $exp\ b\ ((\mathbf{1}_+\ m)+n)$ 中的 $\mathbf{1}_+$ 往外提，將 $exp\ b$ 往內側推，使得式子中出現 $exp\ b\ (m+n)$。一旦成功，就可運用歸納假設，將其改寫成 $exp\ b\ m \times exp\ b\ n$！接下的就是機械化地收尾、將式子整理成 $exp\ b\ (\mathbf{1}_+\ m) \times exp\ b\ n$ 了。

這呼應到第 0.2節所說的「讓符號為你工作」。光從語意上想，我們不易理解為何 $exp\ b\ ((\mathbf{1}_+\ m)+n)$ 能夠等於 $exp\ b\ (\mathbf{1}_+\ m) \times exp\ b\ n$. 但符號給我們線索。我們可觀察式子的結構，暫時不去想語意；我們的目標是操作、移動這些符號，將它們轉換成可使用歸納假設的形式。因此，接下來的演算推導便有所依據而非盲目進行：已知目標是把某符號往外提或往內推，我們就可尋找、使用可達到此目的的規則。這些規則包括已知函數的定義、或諸如分配律、遞移律、結合律等等數學性質。若很明顯地缺了一個想要的性質，也許可以把它當作引理另外證證看。符號幫助我們，使我們的思考清晰而有方向。

習題 2.1 — 證明 $1 \times k = k$. 這個證明並不需要歸納。
習題 2.2 — 證明 $(+)$ 之結合律: $m+(n+k) = (m+n)+k$. 此證明中你使用的述語是什麼？
習題 2.3 — 證明 $k+1 = k$. 你需要使用歸納法嗎？用什麼述語？

最後，說到自然數上的歸納定義，似乎不得不提階層 (factorial)。用非正式的寫法，$fact\ n = n \times (n-1) \times (n-2) \times \ldots \times 1$. 形式化的定義如下：

$$
\begin{aligned}
&fact :: \mathbb{N} \to \mathbb{N} \\
&fact\ \mathbf{0} \qquad = 1 \\
&fact\ (\mathbf{1}_+\ n) = (\mathbf{1}_+\ n) \times fact\ n \quad .
\end{aligned}
$$

我們在定理 2.5 中將會談到階層與排列的關係。

2.4 串列與其歸納定義

如同第 1.8所述，「元素型別為 a的串列」可定義成如下的資料型別：[5]

$$\mathbf{data}\ \mathsf{List}\ a = [\,] \mid a : \mathsf{List}\ a \quad .$$

這個定義可以理解為

1. $[\,]$ 是一個串列，
2. 若 xs 是一個元素型別為 a的串列，x 型別為 a, 則 $x : xs$ 也是一個元素型別為 a的串列，
3. 此外沒有其他元素型別為 a的串列。

我們不難發現 $\mathsf{List}\ a$ 和 $\mathbb{N}$ 是相當類似的資料結構：$[\,]$ 相當於 $\mathbf{0}$, $(:)$ 則類似 $\mathbf{1}_+$, 只是此處我們不只「加一」，添加的那個東西多了一些資訊，是一個型別為 a的元素。或著我們可反過來說，串列「只是」在每個 $\mathbf{1}_+$上都添了一些資訊的自然數！既然自然數與串列有類似的結構，不難想像許多自然上的函數、自然數的性質，都有串列上的類似版本，

2.4.1 串列上之歸納定義

和自然數類似，許多串列上的函數可歸納地定義出來。由於串列只可能由 $[\,]$或 $(:)$做出，定義串列上的函數時也分別處理這兩個情況。基底情況為 $[\,]$, 而欲定義 $f\ (x : xs)$ 的值時，可假設 $f\ xs$ 已算出來了：

$$
\begin{aligned}
&f :: \mathsf{List}\ a \to b \\
&f\ [\,] \qquad = e \\
&f\ (x : xs) = \ldots f\ xs \ldots
\end{aligned}
$$

來看些例子吧！「算一個陣列的和」可能是許多人學到陣列後得寫的頭幾個練習程式。串列版的和可以這麼寫：

$$
\begin{aligned}
&sum :: \mathsf{List\ Int} \to \mathsf{Int} \\
&sum\ [\,] \qquad = 0 \\
&sum\ (x : xs) = x + sum\ xs \quad .
\end{aligned}
$$

基底狀況中，空串列的和應是 0。歸納步驟中，我們要算 $x : xs$ 的和，可假設我們已算出 xs 的和，再加上 x 即可。計算串列長度的 $length$ 有很類似的定義：

$$
\begin{aligned}
&length :: \mathsf{List}\ a \to \mathbb{N} \\
&length\ [\,] \qquad = \mathbf{0} \\
&length\ (x : xs) = \mathbf{1}_+\ (length\ xs) \quad .
\end{aligned}
$$

[5] Haskell 中「元素型別為 a的串列」寫成 $[a]$. 由於這樣的符號在教學中遇到許多困難，本書中寫成 $\mathsf{List}\ a$.

在歸納步驟中，我們想計算 $x : xs$ 的長度，只需假設我們已知 xs 的長度，然後加一。事實上，*length* 剛好體現了前述「List a 只是在每個 $\mathbf{1}_+$ 上添了資訊的自然數」一事：*length* 把串列走過一遍，將 $[\,]$ 代換成 $\mathbf{0}$，並將每個 $(\_:)$ 中附加的資訊拋棄，代換成 $\mathbf{1}_+$。

函數 $map\, f :: \mathsf{List}\, a \to \mathsf{List}\, b$，也就是 *map* 給定函數 f 的結果，也可在串列上歸納定義：

$$\begin{aligned} &map :: (a \to b) \to \mathsf{List}\, a \to \mathsf{List}\, b \\ &map\, f\, [\,] \quad\;\; = [\,] \\ &map\, f\, (x : xs) = f\, x : map\, f\, xs \;\; . \end{aligned}$$

基底狀況的合理結果是 $[\,]$. 歸納步驟中，要對 $x : xs$ 中的每個元素都做 f, 我們可假設已經知道如何對 xs 中的每個元素都做 f, 把其結果接上 $f\, x$ 即可。

函數 $(+\!\!+)$ 把兩個串列接起來。如果我們在其左邊的參數上做歸納定義，可得到：[6]

$$\begin{aligned} &(+\!\!+) :: \mathsf{List}\, a \to \mathsf{List}\, a \to \mathsf{List}\, a \\ &[\,] \quad\;\; +\!\!+ ys = ys \\ &(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \;\; . \end{aligned}$$

空串列接上 ys 仍是 ys. 歸納步驟中，要把 $x : xs$ 接上 ys, 我們可假設已有辦法把 xs 接上 ys, 然後只需添上 x 即可。

請讀者比較一下 $(+\!\!+)$與自然數加法 $(+)$的定義，會發現兩者的結構一模一樣！如果串列是在每個 $\mathbf{1}_+$ 中加上資料的自然數，$(+\!\!+)$就是串列上的加法了。若要形式化地把 List a, $\mathbb{N}$, $(+\!\!+)$, 與 $(+)$ 牽上關係，連接他們的橋樑就是 *length* — $xs +\!\!+ ys$ 的長度，應是 xs 與 ys 的長度之和！意即：

$$length\, (xs +\!\!+ ys) = length\, xs + length\, ys \;\; . \tag{2.1}$$

習題 2.10 中將證明此性質。

最後，$(+\!\!+)$ 是反覆使用 $(:)$, 函數 *concat* 則是反覆使用 $(+\!\!+)$:

$$\begin{aligned} &concat :: \mathsf{List}\, (\mathsf{List}\, a) \to \mathsf{List}\, a \\ &concat\, [\,] = [\,] \\ &concat\, (xs : xss) = xs +\!\!+ concat\, xss \;\; . \end{aligned}$$

2.4.2 串列上之歸納證明

如果 List a 是一個歸納定義出的資料結構，我們應可以在 List a 之上做歸納證明。確實，串列上的歸納法可寫成：

$$\textbf{串列上之歸納法：}\;\; (\forall xs \cdot \mathsf{P}\, xs) \;\Leftarrow\; \mathsf{P}\, [\,] \wedge (\forall x\, xs \cdot \mathsf{P}\, (x : xs) \Leftarrow \mathsf{P}\, xs) \;\; .$$

以文字敘述的話：給定一個述語 $\mathsf{P} :: \mathsf{List}\, a \to \mathsf{Bool}$, 若要證明 $\mathsf{P}\, xs$ 對所有 xs 都成立，只需證明 $\mathsf{P}\, [\,]$ 和「對所有 x 和 xs, 若 $\mathsf{P}\, xs$ 則 $\mathsf{P}\, (x : xs)$」。

[6] 依照 Haskell 的運算元優先順序，$x : (xs +\!\!+ ys)$ 其實可寫成 $x : xs +\!\!+ ys$, 一般也常如此寫。此處為了清楚而加上括號。

下述的 *map* 融合定理 (*map-fusion theorem*) 是關於 *map* 極常用的定理之一。所謂「融合」在此處是把兩個 *map* 融合為一。我們日後會見到更多的融合定理。

定理 2.2 (*map* 融合定理). 對任何 f 與 g, $map\, f \cdot map\, g = map\, (f \cdot g)$.

作為一個例子，我們試著證明定理 2.2。我們目前只會用歸納證明，但是 $map\, f \cdot map\, g = map\, (f \cdot g)$ 的左右邊都是函數，沒有出現串列也沒有出現自然數。該拿什麼東西來歸納呢？

回顧外延相等（定義 1.8）：當 h, k 均是函數，$h = k$ 的意思是對任何參數 x, $h\, x = k\, x$. 因此，將待證式左右邊各補上參數，並將 $(\cdot)$ 展開，可得知其意義為對任何 xs,

$$map\, f\, (map\, g\, xs) = map\, (f \cdot g)\, xs \quad .$$

我們便可以在 xs上做歸納了！

Proof. 當 $xs := [\,]$，等式兩邊皆歸約為 $[\,]$. 考慮 $xs := x : xs$ 的情況：

$$\begin{aligned}
& map\, f\, (map\, g\, (x : xs)) \\
= \quad & \{\, map \text{ 之定義} \,\} \\
& map\, f\, (g\, x : map\, g\, xs) \\
= \quad & \{\, map \text{ 之定義} \,\} \\
& f\, (g\, x) : map\, f\, (map\, g\, xs) \\
= \quad & \{\, (\cdot) \text{ 之定義} \,\} \\
& (f \cdot g)\, x : map\, f\, (map\, g\, xs) \\
= \quad & \{\, \text{歸納假設} \,\} \\
& (f \cdot g)\, x : map\, (f \cdot g)\, xs \\
= \quad & \{\, map \text{ 之定義} \,\} \\
& map\, (f \cdot g)\, (x : xs) \quad .
\end{aligned}$$

□

習題 2.4 — 證明對所有 xs, $xs +\!\!+ [\,] = xs$. 比較本題與習題 2.3的證明。

習題 2.5 — 定義函數 $reverse :: \mathsf{List}\ a \to \mathsf{List}\ a$, 將輸入的串列反轉。例如 $reverse\, [1,2,3,4,5] = [5,4,3,2,1]$.

習題 2.6 — 證明對所有 f, $length \cdot map\, f = length$.

習題 2.7 — 證明對所有 x, $sum \cdot map\, (x\times) = (x\times) \cdot sum$.

習題 2.8 — 證明對所有 xs, $sum\, (map\, (\mathbf{1}_+)\, xs) = length\, xs + sum\, xs$.

習題 2.9 — 證明對所有 xs 與 y, $sum\, (map\, (const\, y)\, xs) = y \times length\, xs$.

討論自然數時，習題 2.2曾請讀者證明加法都滿足結合律。此處示範證明類似定理的串列版：

定理 2.3. $(+\!\!+)$ 滿足結合律。意即，對任何 xs, ys, 和 zs, $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$.

等式證明的步驟該多詳細？

本書中目前為止的等式證明相當細：每一個定義展開都成為獨立的步驟。這是為了教學目的，實務上不一定得如此。以我而言，自己的研究手稿中可能會將步驟寫得極詳細，為確保每個細節正確，並讓他人（或幾年後已經忘記細節的自己）在不需知道上下文的情況下也能機械化地檢查每個步驟。但在論文中，因篇幅有限，及考量讀者一次能處理的資訊量有限，發表出的證明可能會省略許多步驟。

實務上，被認為簡單、不寫出也不妨礙理解的步驟或說明都可被省略。但何謂簡單則很依靠作者的判斷與習慣。一般說來，僅展開定義的步驟用電腦便可自動做到，通常是可精簡掉的。最好寫出的步驟則可能是決定整個證明之結構的、不易以電腦決定而得靠人類智慧與經驗的，等等。這可能包括使用歸納假設的那步，或使用較特別的引理時。例如，性質 2.1的歸納步驟證明可能被精簡如下：

$$
\begin{aligned}
& \mathit{length}\,((x : xs) +\!\!+ ys) \\
=\ & \mathbf{1}_+ \,(\mathit{length}\,(xs +\!\!+ ys)) \\
=\ & \quad \{\, \text{歸納假設} \,\} \\
& \mathbf{1}_+ \,(\mathit{length}\, xs + \mathit{length}\, ys) \\
=\ & \mathit{length}\,(x : xs) + \mathit{length}\, ys \ .
\end{aligned}
$$

Proof. 上述式子中有三個變數，我們怎麼得知該在哪一個變數上做歸納呢？此時絕對別急著把三個變數都拆開，變成多達八種狀況。觀察：如果要歸約等號左邊的 $(xs +\!\!+ ys) +\!\!+ zs$，根據 $(+\!\!+)$ 的定義，得對 $xs +\!\!+ ys$ 做狀況分析；要歸約 $xs +\!\!+ ys$，又得對 xs 做狀況分析。同樣地，根據 $(+\!\!+)$ 的定義，要歸約等號右邊的 $xs +\!\!+ (ys +\!\!+ zs)$ 得對 xs 做狀況分析。不論左右邊，最關鍵的值都是 xs. 因此我們在 xs 之上做歸納。

狀況 $xs := []$:

$$
\begin{aligned}
& ([] +\!\!+ ys) +\!\!+ zs \\
=\ & \quad \{\, (+\!\!+) \text{ 之定義} \,\} \\
& ys +\!\!+ zs \\
=\ & \quad \{\, (+\!\!+) \text{ 之定義} \,\} \\
& [] +\!\!+ (ys +\!\!+ zs) \ .
\end{aligned}
$$

狀況 $xs := x : xs$:

$$
\begin{aligned}
& ((x : xs) +\!\!+ yz) +\!\!+ zs \\
=\ & \quad \{\, (+\!\!+) \text{ 之定義} \,\} \\
& (x : (xs +\!\!+ ys)) +\!\!+ zs \\
=\ & \quad \{\, (+\!\!+) \text{ 之定義} \,\} \\
& x : ((xs +\!\!+ ys) +\!\!+ zs) \\
=\ & \quad \{\, \text{歸納假設} \,\} \\
& x : (xs +\!\!+ (ys +\!\!+ zs)) \\
=\ & \quad \{\, (+\!\!+) \text{ 之定義} \,\} \\
& (x : xs) +\!\!+ (ys +\!\!+ zs) \ .
\end{aligned}
$$

□

基底狀況的證明很簡單。至於歸納步驟，同樣地，前兩步都是為了湊出 $(xs +\!\!+ ys) +\!\!+ zs$, 以便使用歸納假設。既然 $(+\!\!+)$ 滿足結合律，日後我們寫 $xs +\!\!+ ys +\!\!+ zs$ 就可不加括號了。

習題 2.10 — 證明性質(2.1)：對所有 xs 與 ys, $length\ (xs +\!\!+ ys) = length\ xs + length\ ys$.

習題 2.11 — 證明對所有 f, xs, 與 ys, $map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys$.

2.5 從資料、程式、到證明

本節再用一個例子談談「讓符號為我們工作」。考慮證明下述性質：

$$sum \cdot concat = sum \cdot map\ sum \quad . \tag{2.2}$$

根據外延相等，這相當於證明對所有 xss, $sum\ (concat\ xss) = sum\ (map\ sum\ xss)$. 當 $xss := [\,]$, 等號兩邊都歸約成 0. 考慮 $xss := xs : xss$ 的情況：

$$\begin{aligned}
& sum\ (concat\ (xs : xss)) \\
= \quad & \{\ concat\ 之定義\ \} \\
& sum\ (xs +\!\!+ concat\ xss) \\
= \quad & \{\ 因\ sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys\ \} \\
& sum\ xs + sum\ (concat\ xss) \\
= \quad & \{\ 歸納假設\ \} \\
& sum\ xs + sum\ (map\ sum\ xss) \\
= \quad & \{\ sum\ 之定義\ \} \\
& sum\ (sum\ xs : map\ sum\ xss) \\
= \quad & \{\ map\ 之定義\ \} \\
& sum\ (map\ sum\ (xs : xss)) \quad .
\end{aligned}$$

讀者對這個證明最大的疑問可能是：我們怎麼知道該用 $sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$ 呢？為什麼當我們遇到 $sum\ (xs +\!\!+ concat\ xss)$, 我們不是把 xs 再拆成首和尾，甚至把 xss 拆開？這是許多綜合考量的結果。首先，我們預期會使用歸納假設，因此證明前幾步的目的均為把 sum 推到 $concat\ xss$ 左側，試圖做出 $sum\ (concat\ xss)$. 此處我們再強調一個觀念：*證明中的步驟不是瞎猜的，而是有目的的*。符號給我們提示：我們需要把 sum 往右推，因此我們試圖尋找能完成這個目標的性質。

其次，*證明的結構跟隨著程式的結構*。由於 $concat$ 是由 $(+\!\!+)$ 定義的，每一個關於 $concat$ 的定理，均很有可能奠基在一個關於 $(+\!\!+)$ 的相對定理上。為了證明一個描述 sum 與 $concat$ 的關係的定理，我們可能會需要一個關於 sum 與 $(+\!\!+)$ 的性質。

有了符號給我們的這兩個線索，我們便可以運用我們對語意的了解：sum 與 $(+\!\!+)$ 應該會滿足什麼性質？我們可猜測應該是 $sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$ — 根據我們對 sum 與 $(+\!\!+)$ 的語意上的理解，這性質成立的機會很大。而且，這個性質允許我們把 sum 推到右邊。

符號仍舊幫助了我們。我們不可能期待所有定理都只靠符號推導出，在關鍵時刻我們仍會需要對於特定領域的、語意上的知識。但符號給了我們許多引導，縮小我們需要猜測的範圍。

至於 $sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$ 該怎麼證明呢？不要急著把 xs 與 ys 都拆開。觀察等號左右邊的式子，根據 $(+\!\!+)$, $(+)$ 與 sum 的定義，兩個式子的歸約都是先對 xs 做情況分析。因此我們可試著在 xs 上做歸納。

證明的結構跟隨著程式的結構 — 這是做證明時相當好用的指引。程式對哪個參數做歸納，我們做證明時就在那個參數上做歸納。函數 f 用函數 g 定義，我們證明 f 的性質時就可以預期會需要一個 g 的相關性質。並非所有證明都可以如此做出，但這個模式在許多情況下都適用。

而如我們所知，程式的結構又跟隨著資料的結構：例如，串列有 $[\,]$ 和 $x : xs$ 兩個情況，我們定義串列上的程式時也分出這兩個情況，定義 $f\ (x : xs)$ 時可以使用 $f\ xs$ 的值。資料結構、程式、與證明於是有著這樣的連續關係：歸納定義出一個資料結構，我們就有了一個在上面歸納定義程式的方法；有了歸納定義的程式，我們便知道怎麼做關於它的歸納證明。一切由資料結構開始。

習題 2.12 — 證明 $length \cdot concat = sum \cdot map\ length$. 我們會需要什麼關於 $(+\!\!+)$ 的相關性質？

習題 2.13 — 證明對所有 f, $map\ f \cdot concat = concat \cdot map\ (map\ f)$. 我們會需要什麼關於 $(+\!\!+)$ 的相關性質？

2.6 更多歸納定義與證明

為讓讀者熟悉，本節中我們多看一些自然數或串列上的歸納定義。

2.6.1 *filter*, *takeWhile*, 與 *dropWhile*

filter 我們曾見過的函數 *filter* 可寫成如下的歸納定義：

$$\begin{aligned} &filter :: (a \to \mathsf{Bool}) \to \mathsf{List}\ a \to \mathsf{List}\ a \\ &filter\ p\ [\,] \quad\ = [\,] \\ &filter\ p\ (x : xs) = \mathbf{if}\ p\ x\ \mathbf{then}\ x : filter\ p\ xs\ \mathbf{else}\ filter\ p\ xs\ \ . \end{aligned}$$

在 *filter* 的許多性質中，我們試著證明下述性質作為例子：

定理 2.4. $filter\ p \cdot map\ f = map\ f \cdot filter\ (p \cdot f)$.

Proof. 和定理 2.2一樣，待證式的左右邊都是函數。根據外延相等，我們將左右邊各補上參數 xs，並在 xs 上做歸納：

$$filter\ p\ (map\ f\ xs) = map\ f\ (filter\ (p \cdot f)\ xs)\ \ .$$

情況 $x := [\,]$ 中左右邊都可化簡成 $[\,]$. 我們看看 $xs := x : xs$ 的情況：

$$\begin{aligned} &filter\ p\ (map\ f\ (x : xs)) \\ =\ &\quad \{ map\ 之定義 \} \end{aligned}$$

$$
\begin{aligned}
& \mathit{filter}\ p\ (f\ x : \mathit{map}\ f\ xs) \\
= \quad & \{\ \mathit{filter}\ \text{之定義}\ \} \\
& \textbf{if}\ p\ (f\ x)\ \textbf{then}\ f\ x : \mathit{filter}\ p\ (\mathit{map}\ f\ xs)\ \textbf{else}\ \mathit{filter}\ p\ (\mathit{map}\ f\ xs) \\
= \quad & \{\ \text{歸納假設}\ \} \\
& \textbf{if}\ p\ (f\ x)\ \textbf{then}\ f\ x : \mathit{map}\ f\ (\mathit{filter}\ (p \cdot f)\ xs)\ \textbf{else}\ \mathit{map}\ f\ (\mathit{filter}\ (p \cdot f)\ xs) \\
= \quad & \{\ \mathit{map}\ \text{之定義}\ \} \\
& \textbf{if}\ p\ (f\ x)\ \textbf{then}\ \mathit{map}\ f\ (x : \mathit{filter}\ (p \cdot f)\ xs)\ \textbf{else}\ \mathit{map}\ f\ (\mathit{filter}\ (p \cdot f)\ xs) \\
= \quad & \{\ \text{因}\ f\ (\textbf{if}\ p\ \textbf{then}\ e_1\ \textbf{else}\ e_2) = \textbf{if}\ p\ \textbf{then}\ f\ e_1\ \textbf{else}\ f\ e_2,\ \text{如後述}\ \} \\
& \mathit{map}\ f\ (\textbf{if}\ p\ (f\ x)\ \textbf{then}\ x : \mathit{filter}\ (p \cdot f)\ xs\ \textbf{else}\ \mathit{filter}\ (p \cdot f)\ xs) \\
= \quad & \{\ \mathit{filter}\ \text{之定義}\ \} \\
& \mathit{map}\ f\ (\mathit{filter}\ (p \cdot f)\ (x : xs))\quad .
\end{aligned}
$$

□

終止與證明 上述證明的倒數第二步為將 $\mathit{map}\ f$ 提到外面，用了一個關於 **if** 的性質：

$$f\ (\textbf{if}\ p\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \ = \ \textbf{if}\ p\ \textbf{then}\ f\ e_1\ \textbf{else}\ f\ e_2 \quad . \tag{2.3}$$

這性質對嗎？若 p 成立，左右手邊都化簡為 $f\ e_1$, 若 p 不成立，左右手邊都化簡為 $f\ e_2$. 因此 (2.3) 應該成立，是嗎？

答案是：如果我們假設的世界中有不終止的程式，(2.3)便不正確了。例如，當 f 是 $\mathit{three}\ x = 3$，而 p 是個永遠執行、不終止的算式（例：$\textbf{let}\ b = \mathit{not}\ b\ \textbf{in}\ b$）：

$$\mathit{three}\ (\textbf{if}\ p\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \ \stackrel{?}{=}\ \textbf{if}\ p\ \textbf{then}\ \mathit{three}\ e_1\ \textbf{else}\ \mathit{three}\ e_2 -\text{”}\ .\text{”} -$$

上述式子的左手邊直接化簡成 3, 但右手邊卻不會終止，因為 **if** 得知道 p 的值。我們找到了 (2.3) 的反例！

在允許可能不終止的程式存在的世界中，(2.3)得多些附加條件。通常的做法是限定 f 須是個*嚴格函數*, 意即 f 的輸入若不終止，f也不會終止。但(2.3)並不是唯一帶著附加條件的性質 — 許多常用性質都得加上類似的附加條件。所有狀況分析也都得將不終止考慮進去，例如，自然數除了 **0** 與 $\mathbf{1}_+\ n$ 之外，還多了第三種情況「不終止」。[7] 推論與證明變得更複雜。有些人因此較喜歡另一條路：藉由種種方法確保我們只寫出會終止的程式，便可假設我們確實活在所有程式都正常終止的世界中。

保護式 v.s. 條件分支 有些人喜歡用保護式語法定義 filter：

$$
\begin{aligned}
& \mathit{filter}\ p\ [\,] \qquad\quad = [\,] \\
& \mathit{filter}\ p\ (x : xs) \mid p\ x \qquad = x : \mathit{filter}\ p\ xs \\
& \qquad\qquad\quad \mid \mathit{otherwise} = \mathit{filter}\ p\ xs \quad .
\end{aligned}
$$

若在此定義下證明定理 2.4，依「證明的結構與程式的結構相同」的原則，順理成章地，我們可在 $xs := x : xs$ 中再分出 $p\ (f\ x)$ 成立與不成立的兩個子狀況：

[7] Bird [1987] 就採用這種作法。

```
狀況 xs := [] : ...
狀況 xs := x : xs : ...
  狀況 p (f x) :
      filter p (map f (x : xs))
    =   { p (f x) 成立 }
      f x : filter p (map f xs)
    = ...  .
  狀況 not (p (f x)) :
      filter p (map f (x : xs))
    =   { (not p (f x)) }
      filter p (map f xs)
    = ...  .
```

這個定義中不用 **if**, 因此證明中也用不上 (2.3), 但該證明要成立仍須假設所有程式都正常終止 — 我們少證了一個「$p\ (f\ x)$ 不終止」的情況（而確實，在此情況下(2.3)並不成立）。喜歡用哪個方式純屬個人偏好。

前幾章提過的 *takeWhile* 與 *dropWhile* 兩函數型別與 *filter* 相同。他們可寫成如下的歸納定義：

$$
\begin{aligned}
&takeWhile :: (a \to \mathsf{Bool}) \to \mathsf{List}\ a \to \mathsf{List}\ a \\
&takeWhile\ p\ [\,] \qquad = [\,] \\
&takeWhile\ p\ (x : xs) = \mathbf{if}\ p\ x\ \mathbf{then}\ x : takeWhile\ p\ xs\ \mathbf{else}\ [\,]\ , \\
&dropWhile :: (a \to \mathsf{Bool}) \to \mathsf{List}\ a \to \mathsf{List}\ a \\
&dropWhile\ p\ [\,] \qquad = [\,] \\
&dropWhile\ p\ (x : xs) = \mathbf{if}\ p\ x\ \mathbf{then}\ dropWhile\ p\ xs\ \mathbf{else}\ x : xs\ \ .
\end{aligned}
$$

兩者都是在輸入串列上做歸納。兩者也都可用保護式語法定義。

習題 2.14 — 證明 $takeWhile\ p\ xs +\!\!+ dropWhile\ p\ xs = xs$.

習題 2.15 — 以保護式語法定義 *takeWhile* 與 *dropWhile*, 以此定義做做看習題 2.14.

2.6.2 *elem* 與不等式證明

不等式證明 給定如下的定義，$elem\ x\ xs$ 判斷 x 是否出現在串列 xs 中：

$$
\begin{aligned}
&elem\ x\ [\,] \qquad = \mathsf{False} \\
&elem\ x\ (y : xs) = (x == y) \vee elem\ x\ xs\ \ .
\end{aligned}
$$

目前為止，我們所練習的都是以 $(=)$將式子串起的等式證明。以下以 *elem*為例，我們嘗試證明一個「不等式」：

$$elem\ z\ xs\ \Rightarrow\ elem\ z\ (xs +\!\!+ ys)\ \ .$$

以口語說出的話：「若 z 出現在 xs 中，z 也出現在 $xs +\!\!+ ys$ 中」。欲證明上式，該從哪一側推到哪一側呢？一般認為從式子較長、或結構較複雜的那側開始，

化簡成較短、較簡單的那側，是較容易的。因此我們嘗試由右側推到左側：由 $elem\ z\ (xs +\!\!+ ys)$ 開始，尋找使之成立的條件，並希望 $elem\ z\ xs$ 是足夠的。

Proof. 在 xs 上做歸納。基底 $xs := [\,]$ 的狀況在此省略，看 $xs := x : xs$ 的狀況：

$$
\begin{aligned}
& elem\ z\ ((x : xs) +\!\!+ ys) \\
\equiv\quad & \{\ (+\!\!+)\ 之定義\ \} \\
& elem\ z\ (x : (xs +\!\!+ ys)) \\
\equiv\quad & \{\ elem\ 之定義\ \} \\
& (z == x) \vee elem\ z\ (xs +\!\!+ ys) \\
\Leftarrow\quad & \{\ 歸納假設\ \} \\
& (z == x) \vee elem\ z\ xs \\
\equiv\quad & \{\ elem\ 之定義\ \} \\
& elem\ z\ (x : xs)\ .
\end{aligned}
$$

□

讀者可注意：第 1, 2, 4 步使用的邏輯關係都是 $(\equiv)$，第 3 步卻是 $(\Leftarrow)$，因此整個證明建立了「若 $elem\ z\ (x : xs)$，則 $elem\ z\ ((x : xs) +\!\!+ ys)$」。

習題 2.16 — 證明 $not\ (elem\ z\ (xs +\!\!+ ys)) \Rightarrow not\ (elem\ z\ xs)$.

習題 2.17 — 證明 $elem\ z\ xs \Rightarrow elem\ z\ (ys +\!\!+ xs)$.

習題 2.18 — 證明 $(\forall x \cdot p\ x \Rightarrow q\ x) \Rightarrow all\ p\ xs \Rightarrow all\ q\ xs$. 其中 all 的定義為：

$$
\begin{aligned}
& all :: (a \to \mathsf{Bool}) \to \mathsf{List}\ a \to \mathsf{Bool} \\
& all\ p\ [\,] \quad\ = \mathsf{True} \\
& all\ p\ (x : xs) = p\ x \wedge all\ p\ xs\ .
\end{aligned}
$$

習題 2.19 — 證明 $all\ (\in xs)\ (filter\ p\ xs)$. 其中 $x \in xs$ 是 $elem\ x\ xs$ 的中序寫法。我們可能需要習題 2.17 和 2.18 的結果，以及下述性質：

$$\mathbf{if}\ p\ \mathbf{then}\ x\ \mathbf{else}\ x = x\ . \tag{2.4}$$

2.6.3 串列區段

前段與後段 本章目前為止討論的歸納定義都依循著這樣的模式：欲定義 $f :: \mathsf{List}\ a \to b$, 只要為 $f\ [\,]$ 與 $f\ (x : xs)$ 找到定義。在定義後者時，只需定義出由 $f\ xs$ 做出 $f\ (x : xs)$ 的關鍵一步。目前為止，這關鍵一步都是加一、加上一個元素等簡單的動作。現在我們來看些更複雜的例子。

例 1.19中曾提及：如果一個串列 xs 可分解為 $ys +\!\!+ zs$, 我們說 ys 是 xs 的一個前段 *(prefix)*, zs 是 xs 的一個後段 *(suffix)*. 例如，串列 $[1,2,3]$ 的前段包括 $[\,]$, $[1]$, $[1,2]$, 與 $[1,2,3]$ （注意：$[\,]$是一個前段，串列 $[1,2,3]$ 本身也是），後段則包括 $[1,2,3]$, $[2,3]$, $[3]$, 與 $[\,]$。我們是否能定義一個函數 $inits :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$, 計算給定串列的所有前段呢？例 1.19給的答案是：

$$inits\ xs = map\ (\lambda n \to take\ n\ xs)\ [0\,..\,length\ xs]\ .$$

如果不用組件，改用歸納定義呢？我們試試看：

$$\begin{aligned} &inits :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &inits\ [\,] \quad\ = ? \\ &inits\ (x : xs) = ? \end{aligned}$$

基底狀況 $inits\ [\,]$ 的可能選擇是 $[[\,]]$（見後述）。至於歸納步驟該怎麼寫？我們用例子來思考。比較 $inits\ [2,3]$ 與 $inits\ [1,2,3]$:

$$\begin{aligned} inits\ [2,3] \quad &= [[\,],[2],[2,3]]\ , \\ inits\ [1,2,3] &= [[\,],[1],[1,2],[1,2,3]]\ . \end{aligned}$$

假設我們已算出 $inits\ [2,3]$, 如何把它加工變成 $inits\ [1,2,3]$? 請讀者暫停一下，思考看看！

一個思路是：如果在 $[[\,],[2],[2,3]]$ 中的每個串列前面都補一個 1, 我們就有了 $[[1],[1,2],[1,2,3]]$. 再和 $inits\ [1,2,3]$ 比較，就只差一個空串列了！因此 $inits$ 的一種定義方式是：

$$\begin{aligned} &inits :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &inits\ [\,] \quad\ = [[\,]] \\ &inits\ (x : xs) = [\,] : map\ (x:)\ (inits\ xs)\ . \end{aligned}$$

在此得提醒：有些讀者認為基底狀況 $inits\ [\,]$ 的值選為 $[[\,]]$，是因為結果的型別是 $\mathsf{List}\ (\mathsf{List}\ a)$（直覺地把每個 List 都對應到一組中括號，或認為 $[[\,]]$ 是型別為 $\mathsf{List}\ (\mathsf{List}\ a)$ 的最簡單的值）。但事實並非如此：畢竟，$[\,]$ 的型別也可以是 $\mathsf{List}\ (\mathsf{List}\ a)$！我們讓 $inits\ [\,] = [[\,]]$ 的原因是空串列 $[\,]$ 的「所有前段」只有一個，恰巧也是 $[\,]$。就如同在自然數上的歸納函數定義中，有些基底狀況是 0, 有些是 1, 有些是別的值，此處我們也依我們的意圖，選定最合適的基底值。

習題 2.20 — 試定義 $inits^+ :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$, 計算一個串列的所有非空前段。例如 $inits^+\ [1,2,3]$ 是 $[[1],[1,2],[1,2,3]]$。當然，其中一個定義方式是 $inits^+ = tail \cdot inits$. 你能以歸納方式定義出 $inits^+$ 嗎？

$$\begin{aligned} inits^+\ [\,] &= ? \\ inits^+\ (x : xs) &= ? \end{aligned}$$

習題 2.21 — 我們驗證一下 $inits$ 在例 1.19 中的組件定義與本章的歸納定義是相等的。定義 $upto :: \mathbb{N} \to \mathsf{List}\ \mathbb{N}$:

$$\begin{aligned} upto\ \mathbf{0} &= [\mathbf{0}] \\ upto\ (\mathbf{1}_+\ n) &= 0 : map\ (\mathbf{1}_+)\ (upto\ n)\ . \end{aligned}$$

使得 $upto\ n = [0\,..\,n]$. 假設 $inits$ 已如本節一般地歸納定義，證明對所有 xs, $inits\ xs = map\ (\lambda n \to take\ n\ xs)\ (upto\ (length\ xs))$. 您可能會需要 map 融合定理（定理 2.2），

定義傳回後段的函數 $tails$ 時可依循類似的想法：如何把 $tails\ [2,3] = [[2,3],[3],[\,]]$ 加工，得到 $tails\ [1,2,3] = [[1,2,3],[2,3],[3],[\,]]$? 這次較簡單：加上 $[1,2,3]$ 即可。的確，串列 $x : xs$ 的後段包括 $x : xs$ 自己，以及 xs 的後段：

$$\begin{aligned} &tails :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &tails\ [\,] = [[\,]] \\ &tails\ (x : xs) = (x : xs) : tails\ xs\ . \end{aligned}$$

在習題 2.44 中我們將證明一個將 $inits$ 與 $tails$ 牽上關係的定理：將 $inits$ 傳回的前段與 $tails$ 傳回的後段照其順序對應，每對接起來都是原來的串列。

連續區段 給定一個串列，許多傳統最佳化問題的目標是計算符合某條件的連續區段（簡稱「區段」）。例如，$[1,2,3]$ 的區段包括 $[\,]$,$[1]$,$[2]$,$[3]$,$[1,2]$,$[2,3]$, 以及 $[1,2,3]$本身。我們可用 $inits$ 與 $tails$ 得到一個串列的所有區段。

$$\begin{aligned} &segments :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &segments = concat \cdot map\ inits \cdot tails\ . \end{aligned}$$

但 $segments$ 無法寫成本章目前這種形式的歸納定義。我們將在以後的章節再討論到 $segments$.

習題 2.22 — 試著把 *segments* 寫成如下的歸納定義：

$$\begin{aligned} &segments :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &segments\ [\,] \quad\ = ? \\ &segments\ (x : xs) = ...segments\ xs... \ , \end{aligned}$$

在歸納步驟中希望由 $segments\ xs$ 湊出 $segments\ (x : xs)$。這是能在不對輸入串列（型別為 $\mathsf{List}\ a$）做任何限制之下做得到的嗎？如果做不到，為什麼？

2.6.4 插入、排列、子串列、與劃分

排列 函數 $fan\ x\ xs$ 把 x 插入 xs 的每一個可能空隙。例如，$fan\ 1\ [2,3,4]$ 可得到 $[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]$. 讀者不妨想想它該怎麼定義？一種可能方式如下：

$$\begin{aligned} &fan :: a \to \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &fan\ x\ [\,] \quad\ = [[x]] \\ &fan\ x\ (y : xs) = (x : y : xs) : map\ (y:)\ (fan\ x\ xs) \ . \end{aligned}$$

有了 *fan*, 我們不難定義 $perms :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$, 計算一個串列所有可能的排列。例如，$perms\ [1,2,3] = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]$：

$$\begin{aligned} &perms :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &perms\ [\,] \quad\ = [[\,]] \\ &perms\ (x : xs) = concat\ (map\ (fan\ x)\ (perms\ xs)) \ . \end{aligned}$$

讀者可思考為何我們需要 *concat*? 如果沒有，會出現什麼錯誤？

基於 *perm* 的這個定義，我們證明一個定理：長度為 n 的串列有 $fact\ n$ 種排列。這個證明將使用到許多輔助性質與引理，有些已經是我們之前證明過的習題，有些則可作為接下來的習題。在本證明之中我們也練習將連續的函數應用 $f\ (g\ (h\ x))$ 寫成函數合成 $f \cdot g \cdot h\ \$\ x$ 以方便計算。

定理 2.5. 對任何 xs, $length\ (perms\ xs) = fact\ (length\ xs)$.

Proof. 在 xs 上做歸納。

基底狀況 $xs := [\,]$:

$$\begin{aligned} &length\ (perms\ [\,]) \\ &= length\ [[\,]] \\ &= 1 \\ &= fact\ (length\ [\,]) \ . \end{aligned}$$

歸納步驟 $xs := x : xs$:

$$\begin{aligned} &length\ (perms\ (x : xs)) \\ &= \quad \{perms, (\cdot), \text{與}\ (\$)\ \text{之定義}\} \end{aligned}$$

$$
\begin{aligned}
& length \cdot concat \cdot map\ (fan\ x) \cdot perms\ \$\ xs \\
= \quad & \{\ 因\ length \cdot concat = sum \cdot map\ length\ (習題\ 2.12),\ map\ 融合\ \} \\
& sum \cdot map\ (length \cdot fan\ x) \cdot perms\ \$\ xs \\
= \quad & \{\ 因\ length \cdot fan\ x = (\mathbf{1}_+) \cdot length\ (習題\ 2.25)\ \} \\
& sum \cdot map\ ((\mathbf{1}_+) \cdot length) \cdot perms\ \$\ xs \\
= \quad & \{\ 因\ map\ length\ (perms\ xs) = map\ (const\ (length\ xs))\ (perms\ xs)\ (習題\ 2.26)\ \} \\
& sum \cdot map\ ((\mathbf{1}_+) \cdot const\ (length\ xs)) \cdot perms\ \$\ xs \\
= \quad & \{\ 因\ sum\ (map\ (\mathbf{1}_+)\ xs) = length\ xs + sum\ xs\ (習題\ 2.8)\ \} \\
& length\ (perms\ xs) + sum\ (map\ (const\ (length\ xs))\ (perms\ xs)) \\
= \quad & \{\ 因\ sum\ (map\ (const\ y)\ xs) = y \times length\ xs\ (習題\ 2.9)\ \} \\
& length\ (perms\ xs) + length\ xs \times length\ (perms\ xs) \\
= \quad & \{\ 四則運算:\ x + y \times x = (1 + y) \times x\ \} \\
& (\mathbf{1}_+\ (length\ xs)) \times length\ (perms\ xs) \\
= \quad & \{\ 歸納假設\ \} \\
& (\mathbf{1}_+\ (length\ xs)) \times fact\ (length\ xs) \\
= \quad & \{\ fact\ 之定義\ \} \\
& fact\ (\mathbf{1}_+\ (length\ xs)) \\
= \quad & \{\ length\ 之定義\ \} \\
& fact\ (length\ (x : xs))\ .
\end{aligned}
$$

□

習題 2.23 — 證明 $map\ f \cdot fan\ x = fan\ (f\ x) \cdot map\ f$.

習題 2.24 — 證明 $perm \cdot map\ f = map\ (map\ f) \cdot perm$.

習題 2.25 — 證明 $length\ (fan\ x\ xs) = \mathbf{1}_+\ (length\ xs)$.

習題 2.26 — 證明 $perms\ xs$ 傳回的每個串列都和 xs 一樣長，也就是 $map\ length\ (perms\ xs) = map\ (const\ (length\ xs))\ (perms\ xs)$. 其中 $const$ 定義於第 30 頁 — $const\ y$ 是一個永遠傳回 y 的函數。

子串列 函數 $sublists :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$ 計算一個串列的所有子串列。後者是類似子集合的概念，只是把順序也考慮進去：ys 是 xs 的子串列，如果將 xs 中的零個或數個元素移除後可得到 ys: 例如 $sublists\ [1,2,3]$ 的結果可能是：$[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]$。怎麼定義 $sublists$ 呢？在基底狀況中，空串列仍有一個子串列 $[]$. 歸納步驟中，$x : xs$ 的子串列可分為兩種：不含 x 的，以及含 x 的。不含 x 的子串列就是 xs 的所有子串列（以下稱作 yss），而含 x 的子串列就是 yss 中的每個串列接上 x. 因此我們可定義：

$$
\begin{aligned}
& sublists :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\
& sublists\ [\,] \qquad = [[\,]] \\
& sublists\ (x : xs) = yss +\!\!+ map\ (x :)\ yss\ \ , \\
& \quad \mathbf{where}\ yss = sublists\ xs\ \ .
\end{aligned}
$$

習題 2.27 — 定義 $splits :: \text{List } a \to \text{List } (\text{List } a \times \text{List } a)$，使 $splits\ xs$ 傳回所有滿足 $ys +\!\!+ zs$ 的 (ys, zs). 例：

$$splits\ [1,2,3] = [([],[1,2,3]),([1],[2,3]),([1,2],[3]),([1,2,3],[])]\ .$$

另一種說法是 $splits\ xs = zip\ (inits\ xs)\ (tails\ xs)$.

習題 2.28 — 證明 $length \cdot sublists = exp\ 2 \cdot length$. 也就是說，長度為 n 的串列的子串列數目為 2^n. 你會需要的性質可能包括 (2.1) ($length\ (xs +\!\!+ ys) = length\ xs + length\ ys$), 以及 $length \cdot map\ f = length$.

劃分　給定串列 $xs :: \text{List } a$. 如果 $yss :: \text{List } (\text{List } a)$ 滿足

1. $concat\ yss = xs$,
2. yss 中的每個串列都不是空的，我們便說 yss 是 xs 的一個劃分 (*partition*)。例如，$[[1],[2,3],[4]]$ 是 $[1,2,3,4]$ 的一個劃分。

我們是否能定義一個函數 $parts :: \text{List } a \to \text{List } (\text{List } (\text{List } a))$, 計算一個串列所有的劃分？可能的方式有很多，其中一種思路如下。首先，空串列 $[]$ 只有一個可能的劃分，即是 $[]$. 考慮 $x : xs$, 如果 xs 已經被劃分完畢，對於 x 我們有兩個選擇：讓它成為單獨的串列，或著將 xs 加入最左邊的串列。寫成歸納定義如下：

$$\begin{aligned}
&parts :: \text{List } a \to \text{List } (\text{List } (\text{List } a)) \\
&parts\ [] \quad\ = [[]] \\
&parts\ (x : xs) = concat\ (map\ (extend\ x)\ (parts\ xs))\ , \\
&\quad \textbf{where } extend\ x\ [] \quad\ = [[x]] \\
&\qquad\qquad extend\ x\ (ys : yss) = [[x] : ys : yss, (x : ys) : yss]\ .
\end{aligned}$$

遞迴呼叫 $parts\ xs$ 找出 xs 的所有劃分，輔助函數 $extend\ x :: \text{List } (\text{List } a) \to \text{List } (\text{List } (\text{List } a))$ 則作用在其中一個劃分上。我們得分出兩個狀況：

1. 如果該劃分含第一個串列 ys 和剩下的 yss, 我們可選擇讓 x 自成一個串列，或著加入 ys 中。我們把這兩種選擇收集到一個串列中，因此 $extend$ 的結果有三層 List。
2. 如果該劃分是空的，加入 x 的劃分必定是 $[[x]]$，但仍需放入一個串列中，表示「只有這一個選擇」。

呼叫 $map\ (extend\ x)$ 的結果需再用一個 $concat$ 聯集在一起。

2.7 再談「讓符號為你工作」

看了這麼多歸納定義，我們做個較複雜的練習，並以此為例再談談「讓符號為你工作」。本節在初次閱讀時可略過。

以下的函數 ins 將一個元素插到一個串列的空隙中。它和第 2.6.4節中的 fan 類似但稍有不同 — $ins\ x\ xs$ 不會把 x 放到 xs 的結尾。[8] 例如：$ins\ 1\ [2,3,4] = [[1,2,3,4],[2,1,3,4],[2,3,1,4]]$。

[8] 本節不使用 fan 而另外定義了 ins 的原因僅是因為如此使得性質 (2.5) 較單純而方便說明。

反函數

假設我們有一個取反函數的運算元：給定 $f :: a \to b$, 其反函數 f^{-1} 的型別是 $b \to \mathsf{List}\ a$。$f^{-1}\ y$ 將所有使得 $f\ x = y$ 的 x 收集在串列中。則 $parts$ 可以有一個更接近本節中文字描述的定義：

$$parts = filter\ (all\ (not \cdot null)) \cdot concat^{-1}\ .$$

計算 $parts\ xs$ 時，我們用 $concat^{-1}$ 找出所有滿足 $concat\ yss = xs$ 的 $\mathsf{List}\ (\mathsf{List}\ a)$, 並且只挑出不含空串列的。

上述定義可以轉換成本節的歸納定義。相關研究可參考 Mu and Bird [2003].

$$\begin{aligned} &ins :: a \to \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &ins\ x\ [\,] \quad\ \ = [\,] \\ &ins\ x\ (y : ys) = (x : y : ys) : map\ (y :)\ (ins\ x\ ys)\ . \end{aligned}$$

請證明對所有 x, ys, 與 zs,

$$map\ (+\!\!+ zs)\ (ins\ x\ ys) +\!\!+ map\ (ys +\!\!+)\ (ins\ x\ zs) = ins\ x\ (ys +\!\!+ zs)\ . \tag{2.5}$$

意即：在 $ys +\!\!+ zs$ 之中插入 x 的所有方法，是 1. 把 x 插入 ys 之中，並把每個結果的右邊接上 zs, 以及 2. 把 x 插入 zs 之中，並把每個結果的左邊接上 ys.

面對一個如此冗長的式子，我們該如何著手？我們可猜想如此複雜的性質大概會需要歸納證明。但在哪個變數上做歸納呢？觀察(2.5)，等號左手邊最外側的運算子是 $(+\!\!+)$。計算 $(+\!\!+)$ 時，根據定義，先被樣式配對的是 $(+\!\!+)$ 左邊的參數，也就是 $map\ (+\!\!+ zs)\ (ins\ x\ ys)$。再來，根據 map 的定義，$map\ (+\!\!+ zs)\ (ins\ x\ ys)$ 的運算要有進展，得先把 $ins\ x\ ys$ 算出來。最後，根據 ins 的定義，計算 $ins\ x\ ys$ 會先拆開 ys. 由此可知，等號左手邊的運算是從拆解 ys 開始的。等號右手邊的情況也一樣：根據 ins 與 $(+\!\!+)$ 的定義，要計算 $ins\ x\ (ys +\!\!+ zs)$，第一個被拆開的是 ys. 因此，欲證明(2.5)，合理的猜測是在 ys 上做歸納！

我們省略 $ys := [\,]$ 的情況，僅考慮歸納情況 $ys := y : ys$。經代換後的式子是 $map\ (+\!\!+ zs)\ (ins\ x\ (y : ys)) +\!\!+ map\ ((y : ys) +\!\!+)\ (ins\ x\ zs)$. 我們採取的策略和證明定理 2.1時一樣：想辦法湊出 $map\ (+\!\!+ zs)\ (ins\ x\ ys) +\!\!+ map\ (ys +\!\!+)\ (ins\ x\ zs)$, 以便使用歸納假設。由此論證如下（步驟加上編號以利說明）：

$$\begin{aligned} & map\ (+\!\!+ zs)\ (ins\ x\ (y : ys)) +\!\!+ map\ ((y : ys) +\!\!+)\ (ins\ x\ zs) \\ = \quad & \{\ 1.\ ins \text{ 與 } map \text{ 之定義}\ \} \\ & (x : y : ys +\!\!+ zs) : map\ (+\!\!+ zs)\ (map\ (y :)\ (ins\ x\ ys)) +\!\!+ map\ ((y : ys) +\!\!+)\ (ins\ x\ zs) \\ = \quad & \{\ 2.\ map \text{ 融合}\ \} \\ & (x : y : ys +\!\!+ zs) : map\ ((+\!\!+ zs) \cdot (y :))\ (ins\ x\ ys) +\!\!+ map\ ((y : ys) +\!\!+)\ (ins\ x\ zs) \\ = \quad & \{\ 3.\ \text{由於 } (+\!\!+ zs) \cdot (y :) = (y :) \cdot (+\!\!+ zs)\ \text{(習題 2.30)}\ \} \\ & (x : y : ys +\!\!+ zs) : map\ ((y :) \cdot (+\!\!+ zs))\ (ins\ x\ ys) +\!\!+ map\ ((y : ys) +\!\!+)\ (ins\ x\ zs) \\ = \quad & \{\ 4.\ map \text{ 融合}\ \} \\ & (x : y : ys +\!\!+ zs) : map\ (y :)\ (map\ (+\!\!+ zs)\ (ins\ x\ ys)) +\!\!+ map\ ((y : ys) +\!\!+)\ (ins\ x\ zs) \end{aligned}$$

$$
\begin{aligned}
&= \quad \{\text{5. 由於 } ((y:ys)+\!\!+) = (y:)\cdot(ys+\!\!+) \text{ (習題 2.29)}\}\\
&\quad (x:y:ys +\!\!+ zs) : map\ (y:)\ (map\ (+\!\!+zs)\ (ins\ x\ ys)) +\!\!+ map\ ((y:)\cdot(ys+\!\!+))\ (ins\ x\ zs)\\
&= \quad \{\text{6. } map \text{ 融合}\}\\
&\quad (x:y:ys +\!\!+ zs) : map\ (y:)\ (map\ (+\!\!+zs)\ (ins\ x\ ys)) +\!\!+ map\ (y:)\ (map\ (ys+\!\!+)\ (ins\ x\ zs))\\
&= \quad \{\text{7. 由於 } map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys\}\\
&\quad (x:y:ys +\!\!+ zs) : map\ (y:)\ (map\ (+\!\!+zs)\ (ins\ x\ ys) +\!\!+ map\ (ys+\!\!+)\ (ins\ x\ zs))\\
&= \quad \{\text{8. 歸納假設}\}\\
&\quad (x:y:ys +\!\!+ zs) : map\ (y:)\ (ins\ x\ (ys +\!\!+ zs))\\
&= \quad \{\text{9. } ins \text{ 與 } (+\!\!+) \text{ 之定義}\}\\
&\quad ins\ x\ ((y:ys) +\!\!+ zs)\ .
\end{aligned}
$$

雖然步驟多、式子長，但策略確定後，整個證明的架構便很清楚。第 1 到 7 步都是為了可在第 8 步使用歸納而做的準備。為湊出 $map\ (+\!\!+zs)\ (ins\ x\ ys) +\!\!+ map\ (ys+\!\!+)\ (ins\ x\ zs)$, 第 2 至第 4 步把 $map\ (+\!\!+zs)$ 往裡推，把第一個 $map\ (y:)$ 往外提；第 5, 6 步則提出另一個 $map\ (y:)$。式子變成 $map\ (y:)\ \ldots +\!\!+ map\ (y:)\ldots$ 的形式後，共同的 $map\ (y:)$ 可在第 7 步提出來。如此，終於可使用歸納假設了。整個證明中，要配對哪個變數、把式子中的哪些項往哪兒移動，都被符號引導著而有線索可循。證明中用到的兩個性質 $(+\!\!+zs)\cdot(y:) = (y:)\cdot(+\!\!+zs)$ 與 $((y:ys)+\!\!+) = (y:)\cdot(ys+\!\!+)$ 並非天外飛來，而是確立了目標後，因為有需求（兩者分別讓我們把 $(+\!\!+zs)$ 與 $(ys+\!\!+)$ 往右推，把 $(y:)$ 往左拉）而設計出並另外證明的性質。

初學者常犯的一個錯誤是：以為所有情況下、所有的變數都需要樣式配對。事實上，過度的樣式配對是形式證明中希望避免的。藉由觀察 (2.5)，我們決定對 ys 做歸納，證明過程中發現如此便已經足夠。如果我們對 ys 和 zs 都做配對，將必須處理四種情況：$ys,zs := [],[],[]$, $ys,zs := [],z:zs$, $ys,zs := [],z:zs$ $ys,zs := y:ys,z:zs$。一個待證明的性質若有 n 個串列，似乎得分出 2^n 種情況 — 只有萬不得已我們才會這麼做。

最嚴重的問題並不是情況數目太多，而是過度的樣式配對破壞了證明的結構。如果對 zs 做配對，許多人會很自然地想把 $ins\ x\ (z:zs)$ 展開。證明可能如此進行：

$$
\begin{aligned}
&\quad map\ (+\!\!+(z:zs))\ (ins\ x\ (y:ys)) +\!\!+ map\ ((y:ys)+\!\!+)\ (ins\ x\ (z:zs))\\
&= \quad \{\text{將 } (+\!\!+) \text{ 的左手邊展開}\}\\
&\quad (x:y:ys +\!\!+ (z:zs)) : map\ (+\!\!+(z:zs))\ (map\ (y:)\ (ins\ x\ ys)) +\!\!+\\
&\qquad map\ ((y:ys)+\!\!+)\ (ins\ x\ (z:zs))\\
&= \quad \{\text{展開 } ins\ x\ (z:zs)\}\\
&\quad (x:y:ys +\!\!+ (z:zs)) : map\ (+\!\!+(z:zs))\ (map\ (y:)\ (ins\ x\ ys)) +\!\!+\\
&\qquad map\ ((y:ys)+\!\!+)\ ((x:z:zs) : map\ (z:)\ (ins\ x\ zs))\\
&= \quad \{map \text{ 的定義}\}\\
&\quad (x:y:ys +\!\!+ (z:zs)) : map\ (+\!\!+(z:zs))\ (map\ (y:)\ (ins\ x\ ys)) +\!\!+\\
&\qquad ((y:ys +\!\!+ (x:z:zs)) : map\ ((y:ys)+\!\!+)\ (map\ (z:)\ (ins\ x\ zs)))\ .
\end{aligned}
$$

要從這樣的式子中再整理出 $map\ (+\!\!+(z:zs))\ (ins\ x\ (y:ys)) +\!\!+ map\ ((y:ys)+\!\!+)\ (ins\ x\ (z:zs))$ 以便做歸納假設，我們得看出如何把（本來不需展開的）$ins\ x\ (z:zs)$ 給收

回來。式子的結構已經被破壞，操作起來的難度便大大提高了。[9]

習題 2.29 — 證明：對所有 y 與 ys, $(y:)\cdot(ys{+\!\!+}) = ((y:ys){+\!\!+})$.

習題 2.30 — 證明：對所有 y 與 zs, $({+\!\!+}zs)\cdot(y:) = (y:)\cdot({+\!\!+}zs)$. 以上兩者都是不需歸納、也不須狀況分析的單純證明。

2.8 其他歸納資料結構

不僅自然數與串列，歸納式的函數定義與證明可用在所有歸納定義的資料結構上。以第 1.10節提及的兩種二元樹為例：

$$\begin{aligned}&\textbf{data}\ \mathsf{ITree}\ a\ \ = \mathsf{Null} \mid \mathsf{Node}\ a\ (\mathsf{ITree}\ a)\ (\mathsf{ITree}\ a)\ \ ,\\ &\textbf{data}\ \mathsf{ETree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a)\ \ .\end{aligned}$$

型別 ETree 可讀解為一個歸納資料結構，我們可在它之上以歸納法定義函數。例如，下述的函數 $minE$ 計算一個 ETree 中最小的元素，$mapE\,f$ 則將函數 f 作用在樹中的每個元素上。

$$\begin{aligned}&minE :: \mathsf{ETree\ Int} \to \mathsf{Int}\\ &minE\ (\mathsf{Tip}\ x)\ \ \ = x\\ &minE\ (\mathsf{Bin}\ t\ u) = minE\ t \downarrow minE\ u\ \ ,\\ &mapE :: (a \to b) \to \mathsf{ETree}\ a \to \mathsf{ETree}\ b\\ &mapE\ f\ (\mathsf{Tip}\ x)\ \ \ = \mathsf{Tip}\ (f\ x)\\ &mapE\ f\ (\mathsf{Bin}\ t\ u) = \mathsf{Bin}\ (mapE\ f\ t)\ (mapE\ f\ u)\ \ .\end{aligned}$$

我們也有如下的歸納原則：

$$\begin{aligned}&\textbf{ETree 上之歸納法：}\ (\forall t :: \mathsf{ETree}\ a \cdot \mathsf{P}\ t)\ \Leftarrow\\ &(\forall x \cdot \mathsf{P}\ (\mathsf{Tip}\ x)) \wedge (\forall t\ u \cdot \mathsf{P}\ (\mathsf{Bin}\ t\ u) \Leftarrow \mathsf{P}\ t \wedge \mathsf{P}\ u)\ \ .\end{aligned}$$

作為例子，我們證明下列性質：

$$minE\ (mapE\ (x+)\ t) = x + minE\ t\ \ .$$

左手邊將樹中的每個元素都加上 x, 然後取最小的。右手邊則告訴我們不必這麼麻煩，先取最小的，再加上 x 即可！

Proof. 基底狀況 $t := \mathsf{Tip}\ y$ 很容易成立。至於歸納狀況 $t := \mathsf{Bin}\ t\ u$, 使其成立的關鍵性質是加法可分配至 $(\downarrow)$ 中：$x + (y \downarrow z) = (x+y) \downarrow (x+z)$:

$$\begin{aligned}&minE\ (mapE\ (x+)\ (\mathsf{Bin}\ t\ u))\\ =\ &\quad \{\ mapE \text{ 與 } minE \text{ 之定義}\ \}\\ &minE\ (mapE\ (x+)\ t) \downarrow minE\ (mapE\ (x+)\ u)\\ =\ &\quad \{\ \text{歸納假設}\ \}\end{aligned}$$

[9] 這其實是我於 2019 年開設的程式語言課程的期末考題。在十數位將 $ins\ x\ (z:zs)$ 展開的同學中，只有一位成功地將它收了回來。

$$
\begin{aligned}
& (x + minE\ t) \downarrow (x + minE\ u) \\
= \quad & \{ \text{算術：加法分配至 } (\downarrow) \text{ 中} \} \\
& x + (minE\ t \downarrow minE\ u) \\
= \quad & \{ minE \text{ 之定義} \} \\
& x + minE\ (\mathsf{Bin}\ t\ u) \ .
\end{aligned}
$$

□

習題 2.31 — 請寫出 ITree 的歸納原則？

習題 2.32 — 試定義函數 $tags :: \mathsf{ITree}\ a \to \mathsf{List}\ a$, 由左至右傳回給定之 ITree 的所有標籤。例如，若

$$
\begin{aligned}
t = \mathsf{Node}\ 3\ (\mathsf{Node}\ 2\ (\mathsf{Node}\ 1\ \mathsf{Null}\ \mathsf{Null})\ \mathsf{Null}) \\
(\mathsf{Node}\ 5\ (\mathsf{Node}\ 4\ \mathsf{Null}\ \mathsf{Null}) \\
(\mathsf{Node}\ 6\ \mathsf{Null}\ \mathsf{Null}))\ ,
\end{aligned}
$$

則 $tags\ t = [1,2,3,4,5,6]$.

習題 2.33 — 試定義函數 $size :: \mathsf{ITree}\ a \to \mathsf{Int}$, 傳回給定的 ITree 之標籤數目 — 例如 $size\ \mathsf{Null} = 0$, $size\ (\mathsf{Node}\ 3\ \mathsf{Null}\ (\mathsf{Node}\ 4\ \mathsf{Null}\ \mathsf{Null})) = 2$.

習題 2.34 — 接續前兩題，證明 $length\ (tags\ t) = size\ t$.

2.9 由集合論看歸納法

歸納證明為何成立？本節試圖以集合論的角度理解歸納法，試圖建立幾個觀念：當我們說某型別是「歸納定義」的，常常代表它是具有某個結構的最小集合。而歸納證明可以理解為證明某型別是所有滿足某性質的事物的子集。本節於初次閱讀時可跳過。

序理論回顧 為了本章的完整性，我們在這兒回顧一些重要定義。對學過序理論或抽象代數的讀者來說，以下概念應不陌生。如果您還不熟悉這些定義，由於它們在程式語言語意中常常使用，值得花些時間弄懂。

定義 2.6 (前序、偏序). 給定集合 S, 令 $(\leqslant)$ 為 S 上的一個二元關係。如果 $(\leqslant)$ 滿足：

- 自反律：對所有 $x \in S$, $x \leqslant x$.
- 遞移律：對所有 $x,y,z \in S$, $x \leqslant y \wedge y \leqslant z \Rightarrow x \leqslant z$,

則 $(\leqslant)$ 被稱為 S 上的一個**前序** *(preorder)*. 如果 $(\leqslant)$ 除上述兩性質外還滿足：

- 反對稱律：對所有 $x,y \in S$, $x \leqslant y \wedge y \leqslant x \Rightarrow x = y$.

則 $(\leqslant)$ 被稱為 S 上的一個**偏序** *(partial order)*. 如果 S 上有偏序 $(\leqslant)$, 我們常把它們放在一起，稱 $(S, (\leqslant))$ 為一個**偏序集合** *(partially ordered set, 或 poset)*.

定義 2.7 (最小上界、最大下界). 給定偏序集合 $(S, (\leqslant))$, 考慮其子集 $T \subseteq S$:

- 如果 $x \in S$ 滿足 $(\forall y \in T : y \leqslant x)$, 則 x 是 T 的一個**上界** *(upper bound)*.

- 如果 T 的所有上界中存在「最小」的，該值稱作 T 的最小上界 (supremum, 或 least upper bound), 記為 $\mathit{sup}\ T$. 依據定義，$\mathit{sup}\ T$ 滿足 $(\forall y \in T : y \leqslant x) \Rightarrow \mathit{sup}\ T \leqslant x$.
- 如果 $x \in S$ 滿足 $(\forall y \in T : y \geqslant x)$, 則 x 是 T 的一個下界 (lower bound).
- 如果 T 的所有下界存在「最大」的，該值稱作 T 的最大下界 (infimum, 或 greatest lower bound), 記為 $\mathit{inf}\ T$. 依據定義，$\mathit{inf}\ T$ 滿足 $(\forall y \in T : y \geqslant x) \Rightarrow \mathit{inf}\ T \geqslant x$.

定義 2.8 (格、完全格). 考慮偏序集合 $(S, (\leqslant))$:

- 如果對任何 $x, y \in S$, $\mathit{sup}\ \{x, y\}$ 和 $\mathit{inf}\ \{x, y\}$ 均存在且都在 S 之中，則 $(S, (\leqslant))$ 是一個格 (*lattice*)。
- 如果對任何 $T \subseteq S$，$\mathit{sup}\ T$ 和 $\mathit{inf}\ T$ 均存在且都在 S 之中，則 $(S, (\leqslant))$ 是一個完全格 (*complete lattice*)。

在本節之中我們只會考慮一種格。令 D 代表所有範式（如 $\mathbf{0}$, $\mathbf{1}_+\ \mathbf{0}$, True, $1 : 2 : [\,]$, $(\lambda x \to x)$...）的集合，我們考慮的格是 $\mathscr{P}D$，即 D 的所有子集形成的集合。其上的偏序就是子集關係 $(\subseteq)$.

定點 再回顧一些與定點相關的理論。

定義 2.9 (定點). 給定完全格 $(\mathsf{A}, (\leqslant))$ 和函數 $f :: \mathsf{A} \to \mathsf{A}$,

1. 如果 $f\ x \leqslant x$, 我們說 x 是 f 的一個前定點 (prefixed point).
2. 如果 $f\ x = x$, 我們說 x 是 f 的一個定點 (fixed point).
3. 如果 $f\ x \geqslant x$, 我們說 x 是 f 的一個後定點 (postfixed point).

定義 2.10 (最小前定點、最大後定點). 給定完全格 $(\mathsf{A}, (\leqslant))$ 和函數 $f :: \mathsf{A} \to \mathsf{A}$,

- 如果 f 的前定點之中存在最小值，我們將它記為 μf. 根據此定義，μf 滿足 $f\ x \leqslant x \Rightarrow \mu f \leqslant x$.
- 如果 f 的後定點之中存在最大值，我們將它記為 νf. 根據此定義，νf 滿足 $x \leqslant f\ x \Rightarrow x \leqslant \nu f$.

定理 2.11. 給定完全格 $(\mathsf{A}, (\leqslant))$ 和函數 $f :: \mathsf{A} \to \mathsf{A}$,

- f 的最小前定點 μf 也是最小的定點，
- f 的最小前定點 νf 也是最大的定點。

歸納定義 回顧自然數的定義：

$$\mathbf{data}\ \mathbb{N} = \mathbf{0} \mid \mathbf{1}_+\ \mathbb{N}\ .$$

第 2.1節對這行定義的解釋是：

1. $\mathbf{0}$ 的型別是 $\mathbb{N}$;
2. 如果 n 的型別是 $\mathbb{N}$, $\mathbf{1}_+\ n$ 的型別也是 $\mathbb{N}$;
3. 此外，沒有其他型別是 $\mathbb{N}$ 的東西。

如果我們把一個型別視作一個集合，上述條件定出了怎麼樣的集合呢？[10] 用 $\mathbb{N}$ 表示我們定出的這個新型別。上述第 1. 點告訴我們 $\mathbf{0}$ 是 $\mathbb{N}$ 的成員，也就是

[10] 請注意：「把型別視為集合」僅在簡單的語意之中成立。本書後來將會討論更複雜的語意，屆時型別並不只是集合。

$\{\mathbf{0}\} \subseteq \mathbb{N}$. 第 2. 點則表示，從 $\mathbb{N}$ 這個集合中取出任一個元素 n, 加上 $\mathbf{1}_+$, 得到的結果仍會在 $\mathbb{N}$ 之中。也就是說 $\{\mathbf{1}_+\, n \mid n \in \mathbb{N}\} \subseteq \mathbb{N}$. 集合基本定理告訴我們 $X \subseteq Z \wedge Y \subseteq Z$ 和 $X \cup Y \subseteq Z$ 是等價的，所以：[11]

$$\{\mathbf{0}\} \cup \{\mathbf{1}_+\, n \mid n \in \mathbb{N}\} \subseteq \mathbb{N} \ . \tag{2.6}$$

意思是說，如果我們定義一個集合到集合的函數 NatF:

$$\mathsf{NatF}\ X = \{\mathbf{0}\} \cup \{\mathbf{1}_+\, n \mid n \in X\} \ ,$$

那麼 (2.6) 可以改寫為

$$\mathsf{NatF}\ \mathbb{N} \subseteq \mathbb{N} \ ,$$

也就是說，$\mathbb{N}$ 是 NatF 的一個前定點！

至於 3. 呢？它告訴我們 $\mathbb{N}$ 僅含恰巧能滿足 1. 和 2. 的元素，沒有多餘。意即，$\mathbb{N}$ 是滿足 *1.* 和 *2.* 的集合之中最小的。若有另一個集合 Z 也滿足 1. 和 2., 我們必定有 $\mathbb{N} \subseteq Z$. 也就是說 $\mathbb{N}$ 是 NatF 的最小前定點：$\mathbb{N} = \mu\,\mathsf{NatF}$!

給定述語 P, 我們把所有「滿足 P 的值形成的集合」也記為 P. [12] 數學歸納法的目的是證明所有自然數都滿足 P。但，「所有自然數都滿足 P」其實就是 $\mathbb{N} \subseteq P$.

如何證明 $\mathbb{N} \subseteq P$? 如前所述，$\mathbb{N}$ 是 NatF 的最小前定點。如果 P 恰巧也是 NatF 的一個前定點，$\mathbb{N} \subseteq P$ 一定得成立。寫成推論如下：

$$\begin{array}{ll}
& \mathbb{N} \subseteq P \\
\Leftarrow & \quad \{\mathbb{N} \text{ 是 NatF 的最小前定點，定義 2.10}\} \\
& \mathsf{NatF}\ P \subseteq P \\
\equiv & \quad \{\mathsf{NatF} \text{ 的定義}\} \\
& \{\mathbf{0}\} \cup \{\mathbf{1}_+\, n \mid n \in P\} \subseteq P \\
\equiv & \quad \{(\cup) \text{ 的泛性質：} X \cup Y \subseteq Z \equiv X \subseteq Z \wedge Y \subseteq Z\} \\
& \{\mathbf{0}\} \subseteq P \ \wedge\ \{\mathbf{1}_+\, n \mid n \in P\} \subseteq P \ .
\end{array}$$

也就是說，如果證出 $\{\mathbf{0}\} \subseteq P$ 和 $\{\mathbf{1}_+\, n \mid n \in P\} \subseteq P$，我們就有 $\mathbb{N} \subseteq P$。其中，

1. $\{\mathbf{0}\} \subseteq P$ 翻成口語便是「P 對 $\mathbf{0}$ 成立」，
2. $\{\mathbf{1}_+\, n \mid n \in P\} \subseteq P$ 則是「若 P 對 n 成立，P 對 $\mathbf{1}_+\, n$ 亦成立」。

正是數學歸納法的兩個前提！

原來，數學歸納法之所以成立，是因為自然數被定義為某函數的最小前定點。事實上，當我們說某型別是「歸納定義」的，意思便是它是某個函數的最小前定點。

以串列為例。為單純起見，先考慮元素皆為 $\mathbb{N}$ 的串列。如下的定義

$$\mathbf{data}\ \mathsf{ListNat} \ = \ [\,] \mid \mathbb{N} : \mathsf{ListNat} \ ,$$

可理解為 $\mathsf{ListNat} = \mu\,\mathsf{ListNatF}$，而 ListNatF 定義為：

[11] $X \subseteq Z \wedge Y \subseteq Z \equiv X \cup Y \subseteq Z$ 被稱為「$(\cup)$ 的泛性質」。

[12] 對任何 A, 函數 $P :: A \to \mathsf{Bool}$ 和「滿足 P 的 A 形成的集合」是同構的。我們可把它們等同視之。

$$\mathsf{ListNatF}\ X = \{[]\} \cup \{n : xs \mid xs \in X, n \in \mathbb{N}\} \ .$$

至於如下定義的、有型別參數的串列，

$$\textbf{data}\ \mathsf{List}\ a\ =\ [] \mid a : \mathsf{List}\ a\ ,$$

則可理解為 $\mathsf{List}\ a = \mu\ (\mathsf{ListF}\ a)$ — $\mathsf{List}\ a$ 是 $\mathsf{ListF}\ a$ 的最小前定點，其中 ListF 定義如下：

$$\mathsf{ListF}\ A\ X = \{[]\} \cup \{x : xs \mid xs \in X, x \in A\} \ .$$

給定某型別 A, 當我們要證明某性質 P 對所有 List A 都成立，實質上是想要證明 $\mathsf{List}\ A \subseteq P$ （同樣地，此處 P 代表所有使述語 P 成立的值之集合）。我們推論如下：

$$\begin{array}{ll} & \mathsf{List}\ A \subseteq P \\ \Leftarrow & \quad \{\ \mathsf{List}\ a = \mu\ (\mathsf{ListF}\ a)\ \} \\ & \mathsf{ListF}\ A\ P \subseteq P \\ \equiv & \quad \{\ \mathsf{ListF}\ \text{之定義}\ \} \\ & \{[]\} \cup \{x : xs \mid xs \in P, x \in A\} \subseteq P \\ \equiv & \quad \{\ (\cup)\ \text{的泛性質}\ \} \\ & \{[]\} \subseteq P\ \wedge\ \{x : xs \mid xs \in P, x \in A\} \subseteq P\ . \end{array}$$

其中 $\{[]\} \subseteq P$ 翻成口語即是「P [] 成立」；$\{x : xs \mid xs \in P, x \in A\} \subseteq P$ 則是「若 P xs 成立，對任何 $x :: A$, P $(x : xs)$ 成立」。

我們之所以能做自然數與串列的歸納證明，因為它們都是歸納定義出的型別 — 它們都被定義成某函數的最小前定點。若非如此，歸納證明就不適用了。那麼，有不是歸納定義出的型別嗎？

讀者可能注意到，本節起初同時談到最小前定點與最大後定點，但後來只討論了前者。事實上，我們也可以把一個資料型別定義為某函數的最大後定點。這時我們說該資料型別是個餘歸納 (*coinductive*) 定義，如此訂出的型別稱為餘資料 (*codata*)。歸納定義出的資料結構是有限的，而餘歸納定義的型別可能包括無限長的資料結構。寫餘資料相關的證明，另有一套稱作餘歸納 (coinduction) 的方法，而餘歸納也影響到我們如何寫與餘資料相關的程式。餘歸納和歸納的相關理論剛好是漂亮的對偶。我們將在 [todo: where] 介紹餘歸納。

習題 2.35 — 回顧第 1.10節中介紹的兩種樹狀結構：

$$\begin{array}{l} \textbf{data}\ \mathsf{ITree}\ a\ = \mathsf{Null} \mid \mathsf{Node}\ a\ (\mathsf{ITree}\ a)\ (\mathsf{ITree}\ a)\ , \\ \textbf{data}\ \mathsf{ETree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a)\ . \end{array}$$

說說看它們分別是什麼函數的最小前定點，並找出它們的歸納原則。

2.10 歸納定義的簡單變化

目前為止，我們所認定「良好」的函數定義是這種形式：

$$f\,\mathbf{0} \quad = \ldots$$
$$f\,(\mathbf{1}_+\,n) = \ldots f\,n\ldots$$

我們知道這樣定義出的函數是個全函數、對所有輸入都會終止、和歸納法有密切關係...。以後的幾個章節中，我們將逐步放鬆限制，允許更有彈性的函數定義模式。我們先從歸納法的一些較簡單的變化開始。

基底狀況的變化 有些函數的值域是「正整數」或「大於 b 的整數」。定義這些函數時我們可用這樣的模式：

$$f_1\,\mathbf{1} \quad = e \qquad\qquad f_b\,b \quad = e$$
$$f_1\,(\mathbf{1}_+\,n) = \ldots f_1\,n\ldots\ , \qquad f_b\,(\mathbf{1}_+\,n) = \ldots f_b\,n\ldots\ .$$

我們可把 f_1 理解為：另外訂了一個資料型別 $\mathbf{data}\ \mathbb{N}^+ = \mathbf{1} \mid \mathbf{1}_+\ \mathbb{N}^+$, 以 $\mathbf{1}$ 為基底狀況，而 f_1 是 $\mathbb{N}^+$ 之上的全函數。f_b 的情況也類似。與使用 $\mathbb{N}$ 的函數混用時，我們就得在這兩個型別之間作轉換。這相當於檢查給 f_1 的輸入都是大於 1 的整數。實務上為了方便，我們仍用同一個型別實作 $\mathbb{N}$ 與 $\mathbb{N}^+$, 就如同實務上用 Int 實作 $\mathbb{N}$ 一樣。

串列的情形也類似。有些函數若能只定義在「不是空的串列」上，其定義會比較合理。對於這些函數，我們可想像有這麼一個「非空串列」資料型別 $\mathbf{data}\ \mathsf{List}^+\ a = [a] \mid a : \mathsf{List}^+\ a$，只是為了方便，我們用普通串列實作它。[13]

例 2.12. 假設 $x{\uparrow}y$ 傳回 x 與 y 中較大的值。函數 $maximum^+$ 傳回一個非空串列中的最大元素：

$$maximum^+ :: \mathsf{List}^+\ \mathsf{Int} \to \mathsf{Int}$$
$$maximum^+\,[x] \quad = x$$
$$maximum^+\,(x : xs) = x{\uparrow}maximum^+\ xs\ .$$

Haskell 標準函式庫中另有一個函數 $maximum :: \mathsf{List}\ \mathsf{Int} \to \mathsf{Int}$, 但該函數需假設 Int 中有一個相當於 $-\infty$ 的值存在，以便當作 $maximum\ [\,]$ 的結果：

$$maximum\ [\,] \quad = -\infty$$
$$maximum\ (x : xs) = x{\uparrow}maximum\ xs\ .$$

例 2.13. 回顧第 *2.6.4*節提及的「劃分 *(partition)*」：如果 $concat\ yss = xs$, 且 yss 中的每個串列都不是空的，則 yss 是 xs 的一個劃分。如果我們限定 xss 為非空串列，計算所有劃分的函數 $parts^+ :: \mathsf{List}^+\ a \to \mathsf{List}\ (\mathsf{List}^+\ (\mathsf{List}^+\ a))$ 可以寫得更簡潔：

$$parts^+\,[x] \quad = [[[x]]]$$
$$parts^+\,(x : xs) = concat\ (map\ (extend\ x)\ (parts^+\ xs))\ ,$$
$$\quad \mathbf{where}\ extend\ x\ (ys : yss) = [[x] : ys : yss, (x : ys) : yss]\ .$$

由於每個劃分一定是非空串列，$extend$ 不需考慮輸入為 $[\,]$ 的情況。

[13] 在一些型別系統更強大的語言中、進行更注重證明的應用時，我確實有將 $\mathbb{N}^+$ 與 List^+ 做成與 $\mathbb{N}$ 和 List 不同的型別的經驗與需求。

多個參數的歸納定義 函數 *take*/*drop* 的功能與 *takeWhile*/*dropWhile* 很類似，但其定義方式卻有些不同：

$$
\begin{aligned}
&take :: \mathbb{N} \to \mathsf{List}\ a \to \mathsf{List}\ a \\
&take\ \mathbf{0}\ \ \ \ \ xs\ \ \ \ \ \ = [\,] \\
&take\ (\mathbf{1}_+\ n)\ [\,]\ \ \ \ \ = [\,] \\
&take\ (\mathbf{1}_+\ n)\ (x : xs) = x : take\ n\ xs\ \ , \\
&drop :: \mathbb{N} \to \mathsf{List}\ a \to \mathsf{List}\ a \\
&drop\ \mathbf{0}\ \ \ \ \ xs\ \ \ \ \ \ = xs \\
&drop\ (\mathbf{1}_+\ n)\ [\,]\ \ \ \ \ = [\,] \\
&drop\ (\mathbf{1}_+\ n)\ (x : xs) = drop\ n\ xs\ \ .
\end{aligned}
$$

我們可把 *take*/*drop* 想成在自然數上歸納定義成的高階函數：$take\ (\mathbf{1}_+\ n)$ 的值是一個 $\mathsf{List}\ a \to \mathsf{List}\ a$ 的函數。定義 $take\ (\mathbf{1}_+\ n)$ 時，我們假設 $take\ n$ 已有定義。唯一的特殊處是我們另分出兩個子情況：串列為 $[\,]$, 或串列為 $x : xs$.

根據「使證明的結構符合程式的結構」的原則，如果要做關於 *take*/*drop* 的證明，一種可能做法是也依循著它們的結構：先拆解自然數，在 $n := \mathbf{1}_+\ n$ 的情況中再分析串列的值。作為例子，我們來驗證第 42頁提到的這個性質：

定理 2.14. 對所有 n 與 xs, $take\ n\ xs +\!\!+ drop\ n\ xs = xs$.

Proof. 在 n 上做歸納。在 $n := [\,]$ 的情況下，等號兩邊都化簡為 $[\,]$. 在 $n := \mathbf{1}_+\ n$ 的情況中，我們再細分出兩種情形：
狀況 $n := \mathbf{1}_+\ n$, $xs := [\,]$. 顯然等號兩邊都化簡為 $[\,]$.
狀況 $n := \mathbf{1}_+\ n$, $xs := x : xs$:

$$
\begin{aligned}
&take\ (\mathbf{1}_+\ n)\ (x : xs) +\!\!+ drop\ (\mathbf{1}_+\ n)\ (x : xs) \\
=\ &\{\ take\ 與\ drop\ 之定義\ \} \\
&x : take\ n\ xs +\!\!+ drop\ n\ xs \\
=\ &\{\ 歸納假設\ \} \\
&x : xs\ \ .
\end{aligned}
$$

□

然而，本章討論的是所有資料結構都是歸納定義、所有函數也都是全函數的世界。如果我們的世界中有不終止程式存在，上式便不見得成立了。

習題 2.36 — 請舉一個在允許不終止程式的 Haskell 中，$take\ n\ xs +\!\!+ drop\ n\ xs = xs$ 不成立的例子。

習題 2.37 — 對任何串列 xs，$head\ xs : tail\ xs = xs$ 都成立嗎？請舉一個反例。

習題 2.38 — 證明對自然數 m, n, $take\ m\ (take\ (m+n)\ xs) = take\ m\ xs$.

習題 2.39 — 證明對自然數 m, n, $drop\ m\ (take\ (m+n)\ xs) = take\ n\ (drop\ m\ xs)$.

習題 2.40 — 證明對自然數 m, n, $drop\ (m+n)\ xs) = drop\ n\ (drop\ m\ xs)$.

函數 zip 是另一個例子。我們可把 $zip\ xs\ ys$ 視為 xs 之上的歸納定義：

$$\begin{aligned}
&zip :: \mathsf{List}\ a \to \mathsf{List}\ b \to \mathsf{List}\ (a \times b) \\
&zip\ [\,]\ \quad ys\ \quad = [\,] \\
&zip\ (x : xs)\ [\,]\ \quad = [\,] \\
&zip\ (x : xs)\ (y : yz) = (x, y) : zip\ xs\ ys\ \ .
\end{aligned}$$

習題 2.41 — 試定義 $zipWith :: (a \to b \to c) \to \mathsf{List}\ a \to \mathsf{List}\ b \to \mathsf{List}\ c$, 並證明 $zipWith\ f\ xs\ ys = map\ (uncurry\ f)\ (zip\ xs\ ys)$.

2.11 完全歸納

說到遞迴定義，費氏數 (Fibonacci number) 是最常見的教科書例子之一。簡而言之，第零個費氏數是 0, 第一個費氏數是 1, 之後的每個費氏數是之前兩個的和。寫成遞迴定義如下：

$$\begin{aligned}
&fib :: \mathbb{N} \to \mathbb{N} \\
&fib\ 0\ \quad\ = 0 \\
&fib\ 1\ \quad\ = 1 \\
&fib\ (2+n) = fib\ (1+n) + fib\ n\ \ .
\end{aligned}$$

但這和我們之前談到的歸納定義稍有不同。我們已知定義 $f\ (\mathbf{1}_+\ n)$ 時可假設 f 在 n 之上已有定義。但在 fib 的定義中，$fib\ (2+n)$ 用到了 fib 的前兩個值。這樣的定義是可以的嗎？

函數 fib 的定義可以視為完全歸納（又稱作強歸納）的例子。回顧：先前介紹的數學歸納法中，使 $\mathsf{P}\ n$ 成立的前提之一是「對所有 n, 若 $\mathsf{P}\ n$ 成立，$\mathsf{P}\ (\mathbf{1}_+\ n)$ 亦成立」。完全歸納則把這個前提增強如下：

給定述語 $\mathsf{P} :: \mathbb{N} \to \mathsf{Bool}$. 若

- 對所有小於 n 的值 i，$\mathsf{P}\ i$ 皆成立，則 $\mathsf{P}\ n$ 亦成立，

則我們可得知 P 對所有自然數皆成立。

以更形式化的方式可寫成：

$$\textbf{完全歸納：}\ (\forall n \cdot \mathsf{P}\ n) \Leftarrow (\forall n \cdot \mathsf{P}\ n \Leftarrow (\forall i < n \cdot \mathsf{P}\ i))\ \ .$$

請注意：前提 $\mathsf{P}\ n \Leftarrow (\forall i < n \cdot \mathsf{P}\ i)$ 隱含 $\mathsf{P}\ 0$ 成立，因為當 $n := 0$, 由於沒有自然數 i 滿足 $i < n$, 算式 $(\forall i < n \cdot \mathsf{P}\ i)$ 可化簡為 True.

在完全歸納法之中，證明 $\mathsf{P}\ n$ 時，我們可假設 P 對所有小於 n 的值都已成立了。對寫程式的人來說，有了完全歸納法，表示我們日後定義自然數上的函數 $f :: \mathbb{N} \to a$ 時，每個 $f\ n$ 都可以自由使用 f 在所有小於 n 的輸入之上的值。因此 $fib\ (2+n)$ 可以用到 $fib\ (1+n)$ 與 $fib\ n$, 因為 $n < 1+n < 2+n$.

例 2.15. 關於完全歸納，離散數學教科書中的一個常見例子是「試證明所有自然數都可寫成不相同的二的乘冪的和」，例如 $50 = 2^5 + 2^4 + 2$. 這可用完全歸納證明。我們以半形式的方式論述如下：令 $\mathsf{P}\, n$ 為「n可寫成一串不相同的二的乘冪的和」. 對所有 n, 我們想要證明 $\mathsf{P}\, n \Leftarrow (\forall i < n \cdot \mathsf{P}\, i)$. 當 n 為 $\mathbf{0}$, 這一串數字即是空串列。當 n 大於零，

- 我們可找到最接近 n 但不超過 n 的二之乘冪，稱之為 m（也就是 $m = 2^k$ 而且 $m \leqslant n < 2 \times m$）.
- 由於 n 不是 $\mathbf{0}$, m 也不是 $\mathbf{0}$. 也因此 $n - m < n$.
- 依據歸納假設 $(\forall i < n \cdot \mathsf{P}\, i)$, 以及 $n - m < n$, $\mathsf{P}\,(n-m)$ 成立 — $n - m$ 可以寫成不相同的二的乘冪的和。
- 也因此 n 可寫成不相同的二的乘冪的和 — 把 $n - m$ 加上 m 即可.

上述證明僅用口語描述，因為如果形式化地寫下這個證明，就等同於寫個將自然數轉成一串二的乘冪的程式，並證明其正確性！下述函數 *binary* 做這樣的轉換，例如，$binary\ 50 = [32, 16, 2]$:

$$
\begin{aligned}
&binary :: \mathbb{N} \to \mathsf{List}\ \mathbb{N} \\
&binary\ \mathbf{0} = [\,] \\
&binary\ n = m : binary\ (n - m)\ \ , \\
&\quad \textbf{where}\ m \quad\ = last\ (takeWhile\ (\leqslant n)\ twos) \\
&\qquad\quad\ twos = iterate\ (2\times)\ 1\ \ .
\end{aligned}
$$

函數 *binary* 是一個完全歸納定義，和上述的證明對應得相當密切：串列 *twos* 是 $[1, 2, 4, 8 \ldots]$ 等等所有二的乘冪，m 是其中最接近而不超過 n 的。遞迴呼叫 $binary\ (n - m)$ 是許可的，因為 $n - m < n$, 而根據完全歸納，我們已假設對所有 $i < n$, $binary\ i$ 皆有定義。

有了完全歸納法，我們在定義自然數上的函數時可允許更靈活的函數定義：

$$
\begin{aligned}
&f :: \mathbb{N} \to a \\
&f\ b = \ldots. \qquad\qquad \{\text{一些基底情況}\} \\
&f\ n = \ldots f\ m \ldots f\ k \ldots \quad \{\text{如果 } m < n \text{ 且 } k < n\}
\end{aligned}
$$

$f\ n$ 的右手邊可以出現不只一個遞迴呼叫，只要參數都小於 n. 但我們必須確定上述定義中的幾個子句足以包含所有狀況，沒有狀況被遺漏。例如，我們若把 *fib* 定義中的 $fib\ 1 = \ldots$ 基底狀況去掉，計算 $fib\ 2 = fib\ 1 + fib\ 0$ 時便會出錯。

習題 2.42 — 證明 $sum\ (binary\ n) = n$.

習題 2.43 — 證明當 $n \geqslant 1$, $fib\ (2 + n) > \alpha^n$, 其中 $\alpha = (1 + \sqrt{5})/2$. 這個證明可用 $n := 1$ 和 $n := 2$ 當基底狀況。

完全歸納 vs 簡單歸納 完全歸納有個較早的稱呼：強歸納 (strong induction)。原本的歸納法則相對被稱呼為簡單歸納或弱歸納。強/弱歸納的稱呼可能使人以為完全歸納比簡單歸納更強 — 意謂前者能證明出一些後者無法證明的定理。事實上，完全歸納與簡單歸納是等價的：能用一個方法證出的定理，用另一個方法也能證出。因此，使用哪一個純粹只是方便性的考量。

反應在程式設計上，給任一個完全歸納定義函數 $f :: \mathbb{N} \to \mathsf{A}$, 我們總能做出一個以簡單歸納定義的函數 $fs :: \mathbb{N} \to \mathsf{List}\ \mathsf{A}$, 滿足 $fs\ n = map\ f\ [n, n-1, ...0]$. 例如，函數 $fibs\ n$ 傳回 $[fib\ n, fib\ (n-1) .., fib\ 0]$.

$$
\begin{aligned}
&fibs :: \mathbb{N} \to \mathsf{List}\ \mathbb{N} \\
&fibs\ 0 \qquad = [0] \\
&fibs\ 1 \qquad = [1,0] \\
&fibs\ (\mathbf{1}_+\ n) = (x_1 + x_0) : x_1 : x_0 : xs \ , \\
&\quad \textbf{where}\ (x_1 : x_0 : xs) = fibs\ n \ .
\end{aligned}
$$

由 fib 到 $fibs$ 的轉換可能令讀者想起演算法中的動態規劃 (*dynamic programming*)。我們將在日後談到這個話題。

串列上的完全歸納 串列與自然數是類似的資料結構。串列上的完全歸納原則便是將 $\mathbf{0}$ 代換為 $[\,]$, 將 $\mathbf{1}_+$ 代換為 $(x:)$. 至於「小於」的關係，可定義為：

$$ys < xs \equiv ys \in tails\ xs \wedge ys \neq xs \ .$$

也就是說 ys 是 xs 的一個後段，但不是 xs 自己。有了如上定義，串列上的完全歸納法是：

$$\textbf{串列的完全歸納：}\ (\forall xs \cdot \mathsf{P}\ xs) \Leftarrow (\forall xs \cdot \mathsf{P}\ xs \Leftarrow (\forall ys < xs \cdot \mathsf{P}\ ys)) \ .$$

應用在編程上，當定義 $f\ (xs:)$ 時，遞迴呼叫可作用在 xs 的任何後段上。

但對許多串列上的函數而言，這樣的模式還不夠靈活。我們得用下一節說到的良基歸納。

2.12 良基歸納

良基歸納 (*well-founded induction*) 可視為完全歸納的再推廣。如果說完全歸納的主角是自然數，使用的是自然數上的「小於 $(<)$」關係，良基歸納則將其推廣到任何型別，使用任一個良基序。

定義 2.16. 給定某型別 A 之上的二元關係 $(\lhd)$。如果從任意一個 $a_0 :: \mathsf{A}$ 開始，均不存在無限多個滿足如下關係的 $a_1, a_2 ...$:

$$... \lhd a_2 \lhd a_1 \lhd a_0 \ ,$$

則 $(\lhd)$ 可稱為一個良基序 (well-founded ordering)。

把 $b \lhd a$ 簡稱為「b 小於 a」。上述定義可以這麼地直覺理解：給定任一個 $a_0 :: \mathsf{A}$，我們找一個滿足 $a_1 \lhd a_0$ 的值 a_1。這種 a_1 可能已不存在，但如果存在，我們再找一個滿足 $a_2 \lhd a_1$ 的 a_2. 說 $(\lhd)$ 是個良基序的意思便是前述過程不可能永遠做下去：總有一天我們得停在一個「最小」的某基底 $a_n :: \mathsf{A}$。

舉例說明：自然數上的「小於 $(<)$」關係是個良基序，但整數上的 $(<)$關係則不是 — 由於負數的存在。實數上的 $(<)$關係也不是。良基序並非得是個全序 (total order)。例如，我們可定義序對上的比較關係如下：

$$(x_1,y_1) \lhd (x_2,y_2) \equiv x_1 < x_2 \wedge y_1 < y_2 \ ,$$

其中 x_1, y_1, x_2, y_2 都是自然數。這麼一來，不論 $(1,4) \lhd (2,3)$ 或 $(2,3) \lhd (1,4)$ 都不成立，但 $(\lhd)$ 仍是個良基序 — 任何兩個自然數形成的序對不論以什麼方式遞減，最晚也得停在 $(0,0)$.

如果 $(\lhd)$ 是個良基序，我們便可在其上做歸納。以直覺來理解的話，如果某函數定義成如此的形式（假設這幾個子句已經包括參數的所有可能情況）：

$$\begin{aligned} &f :: \mathsf{A} \to \mathsf{B} \\ &f\ b = ... && \{ \text{一些基底情況} \} \\ &f\ x = ...f\ y ...f\ z ... && \{ \text{如果 } y \lhd x \text{ 且 } z \lhd x \} \end{aligned}$$

由任何 $f\ x$ 開始，若 x 不是基底情況之一，我們需遞迴呼叫 $f\ y$ 和 $f\ z$。但 y 和 z 在 $(\lhd)$ 這個序上比 x「小」了一點。此後即使再做遞迴呼叫，每次使用的參數又更小了一點。而由於 $(\lhd)$ 是良基序，f 的參數不可能永遠「小」下去 — f 非得停在某個基底情況不可。因此 f 必須正常終止。同樣的原則也用在證明上：

> 給定述語 $\mathsf{P} :: \mathsf{A} \to \mathsf{Bool}$ 以及 A 之上的良基序 $(\lhd)$。若
>
> - 對所有滿足 $y \lhd x$ 的值 y，$\mathsf{P}\ y$ 皆成立，則 $\mathsf{P}\ x$ 亦成立，
>
> 則我們可得知 P 對所有 A 皆成立。

或著可寫成如下形式：

$$\text{良基歸納：}\ (\forall x \cdot \mathsf{P}\ x) \Leftarrow (\forall x \cdot \mathsf{P}\ x \Leftarrow (\forall y \lhd x \cdot \mathsf{P}\ y)) \ ,$$
$$\text{其中 } (\lhd) \text{ 為一個良基序。}$$

終止證明與良基歸納 我們已在許多地方強調：確定程式正常終止是很重要的。我們也知道以簡單歸納與完全歸納定義出的程式均是會正常終止的。但這兩種歸納定義的限制很多。雖然我們已舉了許多例子，仍有些程式難以套入它們所要求的模板中。相較之下，良基歸納寬鬆許多。大部分我們已知、會終止的程式都可視為良基歸納定義。

或著，上述段落應該反過來說。在函數程設中，欲證明某個遞迴定義的函數會終止，最常見的方式是證明該函數每次遞迴呼叫時使用的參數都在某個度量上「變小」了，而這個度量又不可能一直變小下去。因此該函數遲早得碰到基底狀況。換句話說，該函數每次遞迴呼叫的參數符合某個良基序；當我們如此證明一個函數會終止，其實就相當於在論證該函數是一個良基歸納定義。在指令式編程中證明某迴圈會終止的做法也類似。最常見的方式是證明該迴圈每多執行一次，某個量值就會變小，而該量值是不可能一直變小的。也就是說這些量值在每趟迴圈執行時的值符合某個良基序。

以下我們將看幾個遞迴定義的例子。請讀者們想想：這些函數總會正常終止嗎？為何？如果它們是良基歸納，使用的良基序是什麼？

例 2.17 (快速排序). 以下是大家熟悉的快速排序 *(quicksort) [Hoare, 1962]:*

$$\begin{aligned} &qsort :: \mathsf{List\ Int} \to \mathsf{List\ Int} \\ &qsort\ [\,] \quad = [\,] \end{aligned}$$

$$
\begin{aligned}
&qsort\ (x:xs) = qsort\ ys +\!\!+ [x] +\!\!+ qsort\ zs\ \ , \\
&\quad \textbf{where}\ (ys,zs) = (filter\ (\leqslant x)\ xs, filter\ (<x)\ xs)\ \ .
\end{aligned}
$$

空串列是已經排序好的。當輸入為非空串列 $x:xs$，我們將 xs 分為小於等於 x 的，以及大於 xs 的，分別遞迴排序，再將結果接在一起。

函數 $qsort$ 會正常終止，因為每次遞迴呼叫時，作為參數的串列都會減少至少一個元素（因為 x 被取出了），而串列的長度又不可能小於 0. 若要稍微形式地談這件事，可從良基歸納的觀點來看。如果定義：

$$
ys \lhd xs \ \equiv\ length\ ys < length\ xs\ \ ,
$$

在 $qsort\ (x:xs)$ 子句中，$ys \lhd xs$ 和 $zs \lhd xs$ 均被滿足，而 $(\lhd)$ 是一個良基序。因此 $qsort$ 是一個奠立在 $(\lhd)$ 之上的良基歸納定義。

例 2.18 (合併排序). 在串列上，合併排序也是很常使用的排序方式。我們在第 *1.9*節中示範過以全麥編程方式寫成、由下往上的合併排序。此處的寫法則更接近大家一般的認知：拿到一個長度為 n 的串列，將之分割為長度大致為 $n\,/\,2$ 的兩段，分別排序之後合併。同樣地，假設我們已有一個函數 $merge :: \mathsf{List\ Int} \to \mathsf{List\ Int} \to \mathsf{List\ Int}$, 如果 xs 與 ys 已經排序好，$merge\ xs\ ys$ 將它們合併為一個排序好的串列。合併排序可寫成：

$$
\begin{aligned}
&msort :: \mathsf{List\ Int} \to \mathsf{List\ Int} \\
&msort\ [\,] \ \ = [\,] \\
&msort\ [x] = [x] \\
&msort\ xs \ \ = merge\ (msort\ ys)\ (msort\ zs)\ \ , \\
&\quad \textbf{where}\ (ys,zs) = (take\ (n\ `div`\ 2)\ xs, drop\ (n\ `div`\ 2)\ xs)\ \ .
\end{aligned}
$$

要論證 $msort$ 會正常終止，或著說，要將 $msort$ 視為一個良基歸納定義，我們可用和例 *2.17*中一樣的良基序 $(\lhd)$.

但此處請讀者小心檢查：在 $msort\ xs$ 子句中，$ys \lhd xs$ 和 $zs \lhd xs$ 有被滿足嗎？當 $length\ xs = n$, 串列 ys 與 zs 的長度分別是 $n\ `div`\ 2$ 和 $n - n\ `div`\ 2$. 當 $length\ xs = 1$ 時，ys 與 zs 的長度分別是... 0 和 1 — zs 並沒有變短！

這是為何我們需要 $msort\ [x]$ 這個子句把 $length\ xs = 1$ 的情況分開處理。如果沒有這個子句，$msort$ 將有可能不終止 — 讀者不妨試試看。

例 2.19 (最大公因數). 歐幾里得 *(Euclid)* 的《幾何原本》成書於西元前三百年，其中描述「計算最大公因數」的做法可能是世界上最古老的演算法。以下函數計算兩個自然數 (m,n) 的最大公因數。如果兩數相等，它們的最大公因數也是自身。若兩數不相等，其最大公因數會是「大數減小數」與「小數」的最大公因數：

$$
\begin{aligned}
&gcd :: (\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \\
&gcd\ (m,n) \mid m == n \quad\ = n \\
&\qquad\quad \mid otherwise = gcd\ ((m \uparrow n) - (m \downarrow n), m \downarrow n)\ \ .
\end{aligned}
$$

這個程式總會正常終止嗎？為什麼？

事實上，若 m 或 n 其中之一為 0, $gcd\ (m,n)$ 是不會終止的 — 讀者不妨也試試看。若 m, n 均為正整數呢？首先我們先確立：如果初始的 m, n 均為正整數，gcd 每次遞迴呼叫拿到的參數也都是正整數 — 確實如此，因為如果 $m>0, n>0$, 且 $m \neq n$, 那麼 $(m \uparrow n)-(m \downarrow n)$ 與 $m \downarrow n$ 都不會是零或負數。接下來我們可論證：如果 m, n 均為正整數，每次遞迴呼叫中，**兩參數的和**都變小了一些。確實：

$$
\begin{aligned}
& (m \uparrow n)-(m \downarrow n)+(m \downarrow n) \\
= & \ m \uparrow n \\
< & \quad \{ m, n \text{ 均為正整數 } \} \\
& \ m+n \ .
\end{aligned}
$$

因此，我們可得知 gcd 在 m, n 均為正整數時會正常終止。如果把 gcd 當作一個良基歸納，我們用了如下的良基序：

$$(m_1, n_1) \lhd (m_2, n_2) \ \equiv \ m_1+n_1<m_2+n_2 \ ,$$

其中 m_1, n_1, m_2, n_2 均為正整數。

例 2.20 (Curried 函數). 下述函數 *interleave* 將兩個參數中的元素交錯放置。例如 $interleave\ [1,2,3]\ [4,5]=[1,4,2,5,3]$.

$$
\begin{aligned}
& interleave :: \mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a \\
& interleave\ [\,] \quad\ ys = ys \\
& interleave\ xs \quad\ [\,] = xs \\
& interleave\ (x : xs)\ ys = x : interleave\ ys\ xs \ .
\end{aligned}
$$

這可視為一個良基歸納定義嗎？若將 *interleave* 做為傳回函數的高階函數看待，我們比較難看出它是定義在什麼良基序上的。但若把 *interleave* 的兩個參數一起考慮，我們不難看出什麼度量在遞迴呼叫後「變小」了：兩個參數長度的和！

凡是遇到像 *interleave* 的 ***curried*** 函數，我們也可考慮它的 ***uncurried*** 版本：

$$
\begin{aligned}
& interleave' :: (\mathsf{List}\ a \times \mathsf{List}\ a) \to \mathsf{List}\ a \\
& interleave'\ ([\,], \quad ys) = ys \\
& interleave'\ (xs, \quad [\,]) = xs \\
& interleave'\ (x : xs, ys) = x : interleave'\ (ys, xs) \ .
\end{aligned}
$$

函數 $interleave'$ 是個良基定義 — 參數中的兩個串列雖然交換位置，但它們長度的總和會變小。也就是說 $interleave'$ 可視為定義在這個良基序上的函數：

$$(xs_1, ys_1) \lhd (xs_2, ys_2) \ \equiv \ length\ xs_1 + length\ ys_1 < length\ xs_2 + length\ ys_2 \ .$$

凡是 $interleave'$ 有的性質，不難找出 *interleave* 的相對應版本；證明 *interleave* 的性質時，可當成是在證明 $interleave'$ 的相對性質。因此我們也會比較寬鬆地說 *interleave* 也是 $(\lhd)$ 之上的良基歸納定義。

例 2.21. 下列函數被稱作「*McCarthy 91* 函數」：

$$
\begin{aligned}
&mc91 :: \mathbb{N} \to \mathbb{N} \\
&mc91\ n \mid n > 100 \quad = n - 10 \\
&\qquad\quad \mid otherwise = mc91\ (mc91\ (n+11)) \ .
\end{aligned}
$$

讀者不妨先猜猜看 $mc91$ 會傳回什麼？答案是，mc 和以下函數是等價的：

$$
\begin{aligned}
&mc91' \mid n > 100 \quad = n - 10 \\
&\qquad\quad \mid otherwise = 91 \ .
\end{aligned}
$$

[todo: finish this.]

2.13 詞典序歸納

我們終於要定義第 1.9節與 2.12節中都提到的「合併」函數：將兩個已排序好的串列合而為一。函數 $merge$ 最自然的寫法可能是：

$$
\begin{aligned}
&merge :: \mathsf{List\ Int} \to \mathsf{List\ Int} \to \mathsf{List\ Int} \\
&merge\ [\,]\ \quad ys\ \quad = ys \\
&merge\ (x:xs)\ [\,]\ \quad = x:xs \\
&merge\ (x:xs)\ (y:ys) = \textbf{if}\ x \leqslant y\ \textbf{then}\ x : merge\ xs\ (y:ys) \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ y : merge\ (x:xs)\ ys \ .
\end{aligned}
$$

如果兩個串列之中有一個為空串列，合併的結果是另一個。如果兩個都不是空串列，我們比較其第一個元素，以便決定將哪個當作合併後的第一個元素。

但，$merge$ 最後一個子句的第一個遞迴呼叫中，$y:ys$ 沒有變短；第二個遞迴呼叫中，$x:xs$ 沒有變短。這種程式會終止嗎？如果會，是哪種歸納定義呢？

一種看法是將 $merge$ 視作和例 2.20中的 $interleave$ 類似的歸納定義：兩個參數的長度和在遞迴呼叫中變小了。另一個可能是將函數 $merge$ 視作詞典序歸納 (*lexicographic induction*) 的例子 — 詞典序歸納也是良基歸納的一個特例。

先介紹詞典序。我們怎麼決定兩個英文單字在詞典中的先後順序呢？通常是先比較其第一個字母，如果第一個字母便分出了大小，就以此大小為準，不論剩下的字母為何。如果第一個字母一樣，便從第二個字母開始比起。若要以形式化的方式寫下詞典序的定義，我們考慮一個較簡單的狀況：如何比較 $x_1\ y_1$ 和 $x_2\ y_2$ 兩個長度均為二的字串？如果 $(<)$ 是比較單一字元大小的順序，我們把兩個字元的詞典序寫成 $(<;<)$，定義如下：

$$x_1 y_1 (<;<) x_2 y_2 \ \equiv\ x_1 < x_2 \ \vee\ (x_1 = x_2 \wedge y_1 < y_2) \ .$$

如前所述，先比較 x_1 與 x_2, 如果相等，再比較 y_1 與 y_2.

我們可以再稍微擴充一些，考慮 x_i 與 y_i 型別不同的情況：

定義 2.22. 給定義在型別 A 之上的序 $(\lhd)$ 和型別 B 之上的序 $(\prec)$, 它們的詞典序 (lexicographic ordering)，寫做 $(\lhd;\prec)$，是 $(\mathsf{A} \times \mathsf{B})$ 上的一個序，定義為：

$$(x_1, y_1)(\lhd;\prec)(x_2, y_2) \ \equiv\ x_1 \lhd x_2 \ \vee\ (x_1 = x_2 \wedge y_1 \prec y_2) \ .$$

上述定義也可擴充到三個、四個... 元素的序對上。此處便不把他們寫出來了。

關於詞典序的有趣性質相當多，此處僅用到下述性質

定理 2.23. 如果 $(\lhd)$ 與 $(\prec)$ 均為良基序，$(\lhd;\prec)$ 也是良基序。

因此，$(\lhd;\prec)$ 也可用來做歸納定義。我們把使用詞典序的良基序歸納稱作「詞典序歸納」。

回頭看 *merge* 的定義。先考慮下述、在第 1.9節中出現的 uncurried 版本：

$$\begin{aligned}
&\mathit{merge}' :: (\mathsf{List\ Int} \times \mathsf{List\ Int}) \to \mathsf{List\ Int} \\
&\mathit{merge}'\ ([\,],\ \ \ ys) \ \ \ = ys \\
&\mathit{merge}'\ (x:xs,[\,]) \ \ \ = x:xs \\
&\mathit{merge}'\ (x:xs,y:ys) = \textbf{if}\ x \leqslant y\ \textbf{then}\ x:\mathit{merge}'\ (xs,y:ys) \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ y:\mathit{merge}'\ (x:xs,ys)\ \ .
\end{aligned}$$

如果 $(\lhd)$ 是比較串列長度的良基序，我們可說 merge' 是在 $(\lhd;\lhd)$ 之上的歸納定義。確實，

- $(xs,y:ys)\ (\lhd;\lhd)\ (x:xs,y:ys)$, 因為 $xs \lhd x:xs$;
- $(x:xs,ys)\ (\lhd;\lhd)\ (x:xs,y:ys)$, 因為 $x:xs = x:xs$ 且 $ys \lhd y:ys$.

至於 *merge* 則是 merge' 的 curried 版本，因此也是定義良好的。

如前所述，函數 *merge* 的定義不一定得看成辭典序歸納 — 它也可和 *interleave* 一樣看成另一種較簡單的良基歸納 — 比較兩參數的長度之和。接下來的例子就得倚靠辭典序歸納了。

例 2.24. 知名的 *Ackermann* 函數 *(Ackermann's function)* 是一個遞增得相當快的函數。

$$\begin{aligned}
&\mathit{ack} :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathit{ack}\ 0 \qquad n \qquad\ = \mathbf{1}_+\ n \\
&\mathit{ack}\ (\mathbf{1}_+\ m)\ 0 \qquad\ = \mathit{ack}\ m\ 1 \\
&\mathit{ack}\ (\mathbf{1}_+\ m)\ (\mathbf{1}_+\ n) = \mathit{ack}\ m\ (\mathit{ack}\ (\mathbf{1}_+\ m)\ n)\ \ .
\end{aligned}$$

該函數定義上的特殊處之一是 $\mathit{ack}\ (\mathbf{1}_+\ m)\ n$ 的結果又被當作 $\mathit{ack}\ m$ 的參數，因此較難以用理解 *interleave* 的方式理解。但它可以視為詞典序 $(<;<)$ 上的歸納：

- $(m,1)\ (<;<)\ (\mathbf{1}_+\ m,0)$，因為 $m < \mathbf{1}_+\ m$;
- $(\mathbf{1}_+\ m,n)\ (<;<)\ (\mathbf{1}_+\ m,\mathbf{1}_+\ n)$，因為 $n < \mathbf{1}_+\ n$;
- $(m,\mathit{ack}\ (\mathbf{1}_+\ m)\ n)\ (<;<)\ (\mathbf{1}_+\ m,\mathbf{1}_+\ n)$，因為 $m < \mathbf{1}_+\ m$.

2.14 交互歸納

許多工作無法由一個函數獨立完成，而需要許多函數彼此呼叫。本章最後談談這類的交互歸納 (mutual induction) 定義。下列函數定義中，*even* 判斷其輸入是否為偶數。第二個子句告訴我們：如果 n 是奇數，$\mathbf{1}_+\ n$ 便是偶數。但如何判斷一個數字是否為奇數？如果 n 是偶數，$\mathbf{1}_+\ n$ 便是奇數：

$$\begin{aligned}
&\mathit{even} :: \mathbb{N} \to \mathsf{Bool} \\
&\mathit{even}\ \mathbf{0} \qquad = \mathsf{True} \\
&\mathit{even}\ (\mathbf{1}_+\ n) = \mathit{odd}\ n\ \ ,
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathit{odd} :: \mathbb{N} \to \mathsf{Bool} \\
&\mathit{odd}\ \mathbf{0} \qquad = \mathsf{False} \\
&\mathit{odd}\ (\mathbf{1}_+\ n) = \mathit{even}\ n\ \ .
\end{aligned}$$

這類彼此呼叫的定義可以視為一整個大定義。為讓讀者習慣，我們先把 $even$ 與 odd 的定義改寫為 λ 算式與 **case**:

$$\begin{aligned} even = \lambda n \to \mathbf{case}\ n\ \mathbf{of} \\ \quad \mathbf{0} \quad \to \mathsf{True} \\ \quad \mathbf{1}_+\ n \to odd\ n\ \ , \end{aligned} \qquad \begin{aligned} odd = \lambda n \to \mathbf{case}\ n\ \mathbf{of} \\ \quad \mathbf{0} \quad \to \mathsf{False} \\ \quad \mathbf{1}_+\ n \to even\ n\ \ . \end{aligned}$$

上述的定義可以合併成一個：$evenOdd$ 是一個序對，其中有兩個函數 $\mathbb{N} \to \mathsf{Bool}$，其中是 $fst\ evenOdd$ 就是 $even$，$snd\ evenOdd$ 就是 odd:

$$\begin{aligned} &evenOdd :: ((\mathbb{N} \to \mathsf{Bool}) \times (\mathbb{N} \to \mathsf{Bool})) \\ &evenOdd = (\lambda n \to \mathbf{case}\ n\ \mathbf{of}\ \ \mathbf{0} \quad \to \mathsf{True} \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{1}_+\ n \to snd\ evenOdd\ n, \\ &\qquad\qquad\quad \lambda n \to \mathbf{case}\ n\ \mathbf{of}\ \ \mathbf{0} \quad \to \mathsf{False} \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{1}_+\ n \to fst\ evenOdd\ n)\ \ . \end{aligned}$$

習題 2.44 — 證明 $all\ (xs ==)\ (zipWith\ (+\!\!+)\ (inits\ xs)\ (tails\ xs))$. (unfinished)

2.15 參考資料

快速排序最初由 Hoare 在 Communications of the ACM 的演算法專欄中發表為兩個獨立的演算法：將陣列分割為大、小兩塊的「演算法 63: PARTITION」[Hoare, 1961b]，以及用前述演算法將陣列排序的「演算法 64: QUICKSORT」[Hoare, 1961c]。至於「演算法 65: FIND」[Hoare, 1961a] 的功能則是：給定 k，尋找一個陣列中第 k 小的元素。該演算法也使用了 PARTITION, 我們現在常把它稱為「快速選擇 (quickselect)」。該專欄要求作者使用 Algol 語言。Algol 支援遞迴，Hoare 也大方地用了遞迴。當時遞迴仍是新觀念，Hoare 在另一篇論文 [Hoare, 1962] 中（以大量文字）描述不用遞迴的作法。

例 2.17中的快速排序常被當作「函數編程（或 Haskell）漂亮又簡潔」的例證：快速排序用其他語言得寫得落落長，用 Haskell 可在兩三行內清楚地寫完。但這樣的比較並不很公平：例 2.17排序的是串列，而拿來比較的對象通常是將陣列做排序的指令式語言程式。快速排序的主要挑戰之一，也是「演算法 63: PARTITION」的重點，是在 $O(n)$的時間、$O(1)$的額外空間之內完成陣列的分塊。這點在串列版本中並沒有（或不須）表達出來。

Manna and McCarthy [1969] Manna and Pnueli [1970]

CHAPTER 3

搜尋樹

本節以搜尋樹為例，示範樹狀結構上的歸納證明。本節於初次閱讀時可跳過。

許多應用需要將資料收集在某種表達「集合」的資料結構中。我們希望能做的操作包括：尋找並判斷某筆資料是否在集合中、加入某筆新資料、刪除某筆資料，等等。許多演算法都仰賴這些操作能有效率地被實作。

我們可以用串列表達一個集合，但在一般的簡單實作中，在串列中尋找與刪除都需要與串列長度成正比的時間。若集合中的資料是能比大小的，使用一個陣列並將元素排序好，可用二元搜尋法在對數時間內尋找某筆資料。但在陣列中加入與刪除元素也都需要與陣列長度成正比的時間挪出或消除空位。

因此，許多針對此類應用的樹狀結構被發展出來。

3.1 二元搜尋樹

二元搜尋樹 (*binary search tree*)是一類樹狀資料結構的統稱。簡單的二元搜尋樹是一種類似 ITree 的結構（為說明方便，我們讓樹中的元素皆為 Int）：

```
data BTree = Nul | Nod BTree Int BTree  ,
```

另有個附加限制：在所有內部節點 Nod $t\ x\ u$ 中，t 之中的所有元素均小於 x, u 之中的所有元素均大於 x. 以下我們將此限制簡稱為「有序性」。給定一個有序的二元搜尋樹 t，下述函數 $search\ k\ t$ 判斷 k 是否出現在 t 之中：

```
search :: Int → BTree → Bool
search k Nul           = False
search k (Nod t x u) | k < x  = search k t
                     | k == x = True
                     | x < k  = search k u  .
```

這是一個歸納定義。由於有序性，每遇到一個 Nod $t\ x\ u$ 節點，我們可比較 k 與 x 的大小，藉以決定往哪一個子樹做搜尋。

如果二元樹是平衡的 — 意指在樹中的每個 Nod $t\ x\ u$ 之中，t 和 u 的元素數目大致相等，或著至少其比例為某個常數，$search\ k\ (\mathsf{Nod}\ t\ x\ u)$ 每次選擇往 t 或 u 搜尋時都會排除一定比例的元素。如此一來，搜尋一個含 n 個元素的樹可在 $O(\log n)$ 的時間內完成。然而，維持「平衡」並不容易，也是各種二元搜尋樹各顯神通之處。

以下我們多給一些定義，以便更形式化地描述有序性。我們假設有個抽象的「集合」型別 Set a. 含單一元素 x 的集合寫成 $\{x\}$; 聯集寫作 $(\cup)$, 並具有結合律、交換律等性質。下述函數 $elems$ 傳回一個樹中的所有元素（類似練習 2.32 中的 $tags$）：

$$
\begin{aligned}
&elems :: \mathsf{BTree} \to \mathsf{Set\ Int} \\
&elems\ \mathsf{Nul} &&= [\,] \\
&elems\ (\mathsf{Nod}\ t\ x\ u) &&= elems\ t \cup \{x\} \cup elems\ u \quad .
\end{aligned}
$$

有序性可表示為下列述語 $sorted$:

$$
\begin{aligned}
&sorted :: \mathsf{BTree} \to \mathsf{Bool} \\
&sorted\ \mathsf{Nul} &&= \mathsf{True} \\
&sorted\ (\mathsf{Nod}\ t\ x\ u) &&= all\ (<x)\ (elems\ t) \wedge all\ (x<)\ (elems\ u) \wedge \\
& && \quad\ sorted\ t \wedge sorted\ u \quad .
\end{aligned}
$$

其中 $all :: (a \to \mathsf{Bool}) \to \mathsf{Set}\ a \to \mathsf{Bool}$ 檢查一個集合中的元素是否均滿足某述語。函數 all 滿足以下性質：

$$
\begin{aligned}
&all\ p\ \emptyset &&= \mathsf{True} \\
&all\ p\ \{x\} &&= p\ x \\
&all\ p\ (s \cup t) &&= all\ p\ s \wedge all\ p\ t \quad .
\end{aligned}
$$

在二元搜尋樹中插入新元素的函數可定義如下：

$$
\begin{aligned}
&insert :: \mathsf{Int} \to \mathsf{BTree} \to \mathsf{BTree} \\
&insert\ k\ \mathsf{Nul} &&= \mathsf{Nod\ Nul}\ k\ \mathsf{Nul} \\
&insert\ k\ (\mathsf{Nod}\ t\ x\ u) &&\mid k < x = \mathsf{Nod}\ (insert\ k\ t)\ x\ u \\
& && \mid k == x = \mathsf{Nod}\ t\ x\ u \\
& && \mid x < k = \mathsf{Nod}\ t\ x\ (insert\ k\ u) \quad .
\end{aligned}
$$

函數 $insert$ 和 $search$ 的結構很類似：都藉由比較 k 與 x 決定該往哪兒插入元素；函數 $search$ 只在碰到 Nul 時才能傳回 False, 函數 $insert\ k$ 也只將 Nod Nul k Nul 放在原本 Nul 出現之處 — 新插入的節點永遠是最邊緣的葉節點。

我們自然會希望 $insert$ 保持有序性：如果 t 是有序的，$insert\ k\ t$ 也應該是。表示為定理如下：

定理 3.1. 對所有 t 與 k, $sorted\ t \Rightarrow sorted\ (insert\ k\ t)$.

Proof. 在 t 之上做歸納證明。基底情況 $t := \mathsf{Nul}$ 之中，$sorted\ (\mathsf{Nod}\ \mathsf{Nul}\ k\ \mathsf{Nul})$ 顯然成立。歸納情況有 $k < x$, $k == x$, $x < k$ 三種。以下我們只列出第一種。
狀況 $t := \mathsf{Nod}\ t\ x\ u$, $k < x$:

$$
\begin{aligned}
& sorted\ (insert\ k\ (\mathsf{Nod}\ t\ x\ u)) \\
\equiv\ & \quad \{\ insert\ 之定義,\ k < x\ \} \\
& sorted\ (\mathsf{Nod}\ (insert\ k\ t)\ x\ u) \\
\equiv\ & \quad \{\ sorted\ 之定義\ \} \\
& all\ (<x)\ (elems\ (insert\ k\ t)) \wedge all\ (x<)\ (elems\ u) \wedge \\
& sorted\ (insert\ k\ t) \wedge sorted\ u \\
\equiv\ & \quad \{\ 練習\ 3.1\text{: } elems\ (insert\ k\ t) = \{k\} \cup elems\ t\ \} \\
& all\ (<x)\ (\{k\} \cup elems\ t) \wedge all\ (x<)\ (elems\ u) \wedge \\
& sorted\ (insert\ k\ t) \wedge sorted\ u \\
\equiv\ & \quad \{\ all\ 之性質\ \} \\
& k < x \wedge all\ (<x)\ (elems\ t) \wedge all\ (x<)\ (elems\ u) \wedge \\
& sorted\ (insert\ k\ t) \wedge sorted\ u \\
\Leftarrow\ & \quad \{\ k < x,\ 歸納假設\ \} \\
& all\ (<x)\ (elems\ t) \wedge all\ (x<)\ (elems\ u) \wedge \\
& sorted\ t \wedge sorted\ u \\
\equiv\ & \quad \{\ sorted\ 之定義\ \} \\
& sorted\ (\mathsf{Nod}\ t\ x\ u)\ .
\end{aligned}
$$

□

函數 *insert* 並不保證做出的樹能平衡 — 恰恰相反，我們一不小心便會做出偏一邊的樹。例如，$insert\ 5\ (insert\ 4\ (insert\ 3\ (insert\ 2\ (insert\ 1\ \mathsf{Nul}))))$ 得到的結果會是：

$$\mathsf{Nod}\ \mathsf{Nul}\ 1\ (\mathsf{Nod}\ \mathsf{Nul}\ 2\ (\mathsf{Nod}\ \mathsf{Nul}\ 3\ (\mathsf{Nod}\ \mathsf{Nul}\ 4\ (\mathsf{Nod}\ \mathsf{Nul}\ 5\ \mathsf{Nul}))))\ .$$

在這個樹中搜尋 5, 和在串列 $[1,2,3,4,5]$ 之中尋找 5 基本上是一樣的。

許多建立在二元搜尋樹的進階資料結構會在插入元素時多做些操作，想辦法維持樹的平衡。我們將在第 3.2節介紹的紅黑樹就是一個例子。

習題 3.1 — 證明對所有 t 與 k, $elems\ (insert\ k\ t) = \{k\} \cup elems\ t$.

3.2 紅黑樹

在二元搜尋樹的基礎上，紅黑樹 (*red-black tree*)多加了個屬性：每個節點都是紅色或黑色之一。表示成 Haskell 資料結構如下：

```
data RBTree = E | R RBTree Int RBTree
            | B RBTree Int RBTree  ,
```

其中 E 為沒有資料的葉節點，R 為紅色內部節點，B 為黑色內部節點 — 葉節點 E 被視為黑色的。定義 **data** Color = Red | Blk, 下述定義的 $color\ t$ 傳回 t 的根部節點的顏色：

$$
\begin{aligned}
&color :: \mathsf{RBTree} \to \mathsf{Color} \\
&color\ \mathsf{E} &&= \mathsf{Blk} \\
&color\ (\mathsf{R}\ \_\ \_\ \_) &&= \mathsf{Red} \\
&color\ (\mathsf{B}\ \_\ \_\ \_) &&= \mathsf{Blk}\ .
\end{aligned}
$$

我們要求紅黑樹滿足下列性質：

1. 紅黑樹也是二元搜尋樹，意即它得是有序的。
2. 從根部開始到每個葉節點 E 的路徑上的黑節點數目均相同。我們說這樣的一棵樹是平衡的。
3. 紅節點的兩個子代都必須是黑色的，黑節點則無此限制。為方便說明，我們把滿足此條件的樹稱為準紅黑樹。
4. 根節點為黑色的。

其中，關於有序性的討論和前一節原則上相同，此節將之省略。我們假設存在某函數 $sorted :: \mathsf{RBTree} \to \mathsf{Bool}$ 判斷一棵紅黑樹是否有序。我們看看其他性質如何形式化。

首先，函數 *bheight* 定義一棵樹的「黑高度」— 所有路徑上黑色節點的最多數目：

$$
\begin{aligned}
&bheight :: \mathsf{RBTree} \to \mathbb{N} \\
&bheight\ \mathsf{E} &&= 0 \\
&bheight\ (\mathsf{R}\ t\ x\ u) &&= bheight\ t \uparrow bheight\ u \\
&bheight\ (\mathsf{B}\ t\ x\ u) &&= 1 + (bheight\ t \uparrow bheight\ u)\ .
\end{aligned}
$$

說一棵樹「平衡」意指每個內節點中，兩個子樹的黑高度均相等。

$$
\begin{aligned}
&balanced :: \mathsf{RBTree} \to \mathsf{Bool} \\
&balanced\ \mathsf{E} &&= \mathsf{True} \\
&balanced\ (\mathsf{R}\ t\ x\ u) &&= bheight\ t == bheight\ u \wedge balanced\ t \wedge balanced\ u \\
&balanced\ (\mathsf{B}\ t\ x\ u) &&= bheight\ t == bheight\ u \wedge balanced\ t \wedge balanced\ u\ .
\end{aligned}
$$

函數 *semiRB* 檢查一棵樹是否為準紅黑 — 紅節點的兩棵子樹均為黑色：

$$
\begin{aligned}
&semiRB :: \mathsf{RBTree} \to \mathsf{Bool} \\
&semiRB\ \mathsf{E} &&= \mathsf{True} \\
&semiRB\ (\mathsf{B}\ t\ x\ u) &&= semiRB\ t \wedge semiRB\ u \\
&semiRB\ (\mathsf{R}\ t\ x\ u) &&= color\ t == \mathsf{Blk} \wedge color\ u == \mathsf{Blk} \wedge semiRB\ t \wedge semiRB\ u
\end{aligned}
$$

最後，如前所述，紅黑樹需滿足 *sorted*, *balanced*, *semiRB*, 並且根節點須為黑色：

$$
\begin{aligned}
&redBlack :: \mathsf{RBTree} \to \mathsf{Bool} \\
&redBlack\ t = sorted\ t \wedge balanced\ t \wedge semiRB\ t \wedge color\ t == \mathsf{Blk}\ .
\end{aligned}
$$

在這樣一棵二元樹中，由於 *balanced* 被滿足，每條路徑上的黑節點數目均相同；由於 *semiRB* 被滿足，紅節點不會連續出現。因此最長路徑之長度不超過最短路徑的兩倍。在這樣的樹中做二元搜尋，總會在 $O(\log n)$ 的時間內找到資料或走到葉節點。

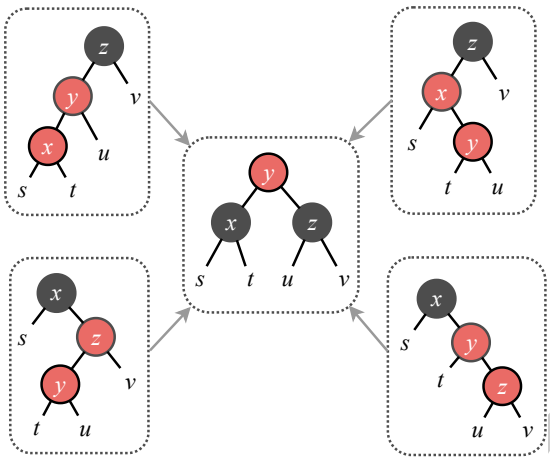


Figure 3.1: 在紅黑樹中插入新元素後做的四種旋轉。

3.2.1 紅黑樹插入

在紅黑樹中插入元素的方式最初和二元搜尋樹相同：一邊搜尋一邊往下走，如果我們碰到 E, 便是插入新元素之處。新元素總在邊緣被插入，而樹的有序性仍保持著。

新加入的節點該是什麼顏色呢？讓新節點為紅色可能是個合理選擇。如此一來，插入新元素不會改變一棵樹的黑高度，也因此如果該樹原本是平衡的，加入新節點後的新樹仍是平衡的。

但這麼做可能破壞準紅黑性質：路徑上有可能出現兩個連續的紅節點。因此我們在插入後的回程途中適時做旋轉，如圖 3.1所示。每當我們看到一個黑節點之下有兩個連續的紅節點，必定是圖中四個角落的四種情形之一（我們只插入了一個元素，最多只有一個多出的紅節點。圖中 s, t, u, 與 v 仍是平衡的準紅黑樹）。我們將每個情形都旋轉成圖中央的情況。如此一來圖中央以 x, z 為根部的兩顆子樹都是黑根部的紅黑樹，y 是紅節點。於是我們再往上檢查，如果又出現四種情況之一就再旋轉，否則往上重建路徑，直到回到根部。如此做出一棵滿足準紅黑性質的樹。最後，如果根部節點為紅色，便直接改成黑色。

我們看看前述的插入演算法可如何表示為 Haskell 程式。我們將主要執行插入與旋轉的函數稱為 $ins :: \mathsf{Int} \to \mathsf{RBTree} \to \mathsf{RBTree}$. 函數 *insert* 則在呼叫 *ins* 之後將樹根改為黑色的：

$$
\begin{aligned}
&\mathit{insert} :: \mathsf{Int} \to \mathsf{RBTree} \to \mathsf{RBTree} \\
&\mathit{insert}\ k\ t = \mathit{blacken}\ (\mathit{ins}\ k\ t)\ \ , \\
&\quad \textbf{where}\ \mathit{blacken}\ (\mathsf{R}\ t\ x\ u) = \mathsf{B}\ t\ x\ u \\
&\qquad\quad \mathit{blacken}\ t = t\ \ .
\end{aligned}
$$

函數 *ins* 在遇上紅節點（$\mathsf{R}\ t\ x\ u$）時和第 3.1中的 *insert* 很類似：比較 k 與 x 以決定該往哪邊插入，並在歸納呼叫後以 R 重做節點。但遇到黑節點 $\mathsf{B}\ t\ x\ u$ 時，我們不使用 B, 而是額外呼叫 *rotate* 函數：

紅黑樹旋轉只需四種情況

本節的的紅黑樹旋轉方式來自 Okasaki [1999]。熟悉資料結構的讀者可能發現它們比一般教科書或網路資源中的處理方式簡單許多：一般資料中常會分出八種以上的狀況，除了主要路徑上的三個節點，也會考慮其兄弟節點的顏色。

Okasaki 發現只需本節的這四種情況就足夠了。讀者也應可發現，採用較簡單的版本，對於證明以及瞭解紅黑樹的性質幫助很大。

那麼，為何一般教科書會用上那麼多情況呢？Okasaki 認為可能是效率考量。一般書中的版本中，有些情況可不需旋轉，只直接改變節點顏色。如此一來需要更動的欄位數目較少。有些情況中轉出的樹根部已是黑色的。在指令式語言中，此時該程序就可以直接結束。

然而，在函數語言中，我們無論如何都需重建整個路徑上的節點。因此上述優點均不明顯。另一方面，Okasaki 也認為此種情況較少的版本是比較適合用於教學中的。

$$
\begin{aligned}
&ins :: \mathsf{Int} \to \mathsf{RBTree} \to \mathsf{RBTree} \\
&ins\ k\ \mathsf{E} = \mathsf{R}\ \mathsf{E}\ k\ \mathsf{E} \\
&ins\ k\ (\mathsf{R}\ t\ x\ u) \mid k < x = \mathsf{R}\ (ins\ k\ t)\ x\ u \\
&\qquad\qquad\quad \mid k == x = \mathsf{R}\ t\ x\ u \\
&\qquad\qquad\quad \mid k > x = \mathsf{R}\ t\ x\ (ins\ k\ u) \\
&ins\ k\ (\mathsf{B}\ t\ x\ u) \mid k < x = rotate\ (ins\ k\ t)\ x\ u \\
&\qquad\qquad\quad \mid k == x = \mathsf{B}\ t\ x\ u \\
&\qquad\qquad\quad \mid k > x = rotate\ t\ x\ (ins\ k\ u)\ .
\end{aligned}
$$

回顧：圖 3.1的四種旋轉都只在根節點為黑色時啟動，因此我們也只在碰上黑節點時呼叫 $rotate$。在 $rotate\ s\ x\ t$ 之中，s 為目前的左子樹，x 為目前的（黑色）節點中的標籤，t 則為目前的右子樹。函數 $rotate$ 的定義如下：

$$
\begin{aligned}
&rotate :: \mathsf{RBTree}\ a \to a \to \mathsf{RBTree}\ a \to \mathsf{RBTree}\ a \\
&rotate\ (\mathsf{R}\ (\mathsf{R}\ s\ x\ t)\ y\ u)\ z\ v = \mathsf{R}\ (\mathsf{B}\ s\ x\ t)\ y\ (\mathsf{B}\ u\ z\ v) \\
&rotate\ (\mathsf{R}\ s\ x\ (\mathsf{R}\ t\ y\ u))\ z\ v = \mathsf{R}\ (\mathsf{B}\ s\ x\ t)\ y\ (\mathsf{B}\ u\ z\ v) \\
&rotate\ s\ x\ (\mathsf{R}\ (\mathsf{R}\ t\ y\ u)\ z\ v) = \mathsf{R}\ (\mathsf{B}\ s\ x\ t)\ y\ (\mathsf{B}\ u\ z\ v) \\
&rotate\ s\ x\ (\mathsf{R}\ t\ y\ (\mathsf{R}\ u\ z\ v)) = \mathsf{R}\ (\mathsf{B}\ s\ x\ t)\ y\ (\mathsf{B}\ u\ z\ v) \\
&rotate\ s\ x\ t = \mathsf{B}\ s\ x\ t\ \ .
\end{aligned}
$$

其中，前四個情況的左邊分別對應到圖 3.1的四個情況；他們的右手邊都是一樣的，對應到圖 3.1正中央的樹。最後的 $rotate\ s\ x\ t$ 則是四種情況之外、不用旋轉的情形。

3.2.2 紅黑樹之性質：高度

許多討論紅黑樹的教材在將插入、刪除等等操作的實作呈現讀者看過之後就結束了，對於其性質的討論意外地不完整。然而，這類資料結構之所以有效，正因為它們需有的性質一直被保持著。談資料結構上的操作卻不證明它們怎麼維護資料結構的性質，可說是缺了最重要的一塊。本章剩下的篇幅中，我們將描述並證明紅黑樹的一些主要性質。

首先我們談談黑高度。讀者稍加嘗試之後會發現，$insert\ k\ t$ 有時會增加 t 的黑高度，有時不會。我們怎知道紅黑樹何時會長高呢？

原來，函數 ins 其實是不會讓樹長高的！我們有如下的定理 — $ins\ k$ 前後樹的黑高度不變：

定理 3.2. 對所有 k 與 t, $bheight\ (ins\ k\ t) = bheight\ t$.

函數 $insert\ k\ t$ 呼叫 $ins\ k\ t$, 得到的樹仍有原來的黑高度。如果 $blacken$ 把樹由紅轉黑，新樹的黑高度才因此加一。否則樹的黑高度仍不變。

系理 3.3. 對所有 k 與 t，如果 $ins\ k\ t$ 為黑色，$bheight\ (insert\ k\ t) = bheight\ t$. 否則 $bheight\ (insert\ k\ t) = 1 + bheight\ t$.

我們將嘗試證明定理 3.2. 回顧我們的原則：證明的結構依循程式的結構。由於 $insert$ 呼叫 ins, 欲證明關於 $insert$ 的系理 3.3，我們需要關於 ins 的定理 3.2. 同樣地，由於 ins 呼叫 $rotate$, 欲證明定理 3.2, 我們需要一個關於 $rotate$ 的引裡：

引理 3.4. 對所有 t, u 與 z, $bheight\ (rotate\ t\ z\ u) = 1 + (bheight\ t \uparrow bheight\ u)$.

這也不意外：$rotate$ 的兩個參數 t 與 u 原本在黑節點之下，旋轉後的黑高度不變，仍是 $1 + (bheight\ t \uparrow bheight\ u)$.

以下我們證明定理 3.2.

Proof. 在 t 之上做歸納。以下只列出幾個代表性狀況。

狀況 $t := \mathsf{E}$.

$$
\begin{aligned}
& bheight\ (ins\ k\ \mathsf{E}) \\
= {} & bheight\ (\mathsf{R}\ \mathsf{E}\ k\ \mathsf{E}) \\
= {} & 0 \\
= {} & bheight\ \mathsf{E}\ .
\end{aligned}
$$

狀況 $t := \mathsf{R}\ t\ x\ u$, $k < x$:

$$
\begin{aligned}
& bheight\ (ins\ k\ (\mathsf{R}\ t\ x\ u)) \\
= {} & \quad \{\ ins\ 之定義；k < x\ \} \\
& bheight\ (\mathsf{R}\ (ins\ k\ t)\ x\ u) \\
= {} & \quad \{\ bheight\ 之定義\ \} \\
& bheight\ (ins\ k\ t) \uparrow bheight\ u \\
= {} & \quad \{\ 歸納假設\ \} \\
& bheight\ t \uparrow bheight\ u \\
= {} & \quad \{\ bheight\ 之定義\ \} \\
& bheight\ (\mathsf{R}\ t\ x\ u)\ .
\end{aligned}
$$

狀況 $t := \mathsf{B}\ t\ x\ u$, $k < x$:

$$
\begin{aligned}
& bheight\ (ins\ k\ (\mathsf{B}\ t\ x\ u)) \\
= {} & \quad \{\ ins\ 之定義；k < x\ \} \\
& bheight\ (balance\ (ins\ k\ t)\ x\ u)
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\text{引理 3.4}\} \\
&\quad 1 + (\mathit{bheight}\ (\mathit{ins}\ k\ t) \uparrow \mathit{bheight}\ u) \\
&= \quad \{\text{歸納假設}\} \\
&\quad 1 + (\mathit{bheight}\ t \uparrow \mathit{bheight}\ u) \\
&= \quad \{\mathit{bheight}\ \text{之定義}\} \\
&\quad \mathit{bheight}\ (\mathsf{B}\ t\ x\ u)\ .
\end{aligned}
$$

□

為求完整，我們也簡述引理 3.4之證明：

Proof. 由於 *rotate* 沒有遞迴、不呼叫其他函數，但有許多狀況，關於 *rotate* 的證明也大都僅是檢查每個情況，雖不難但可能很繁瑣。以下只舉一種狀況為例。

狀況 $(t,z,u) := (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v, z, w)$:

$$
\begin{aligned}
&\quad \mathit{bheight}\ (\mathit{rotate}\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v)\ z\ w) \\
&= \quad \{\mathit{rotate}\ \text{之定義}\} \\
&\quad \mathit{bheight}\ (\mathsf{R}\ (\mathsf{B}\ t\ x\ u)\ y\ (\mathsf{B}\ v\ z\ w)) \\
&= \quad \{\mathit{bheight}\ \text{之定義}\} \\
&\quad (1 + (\mathit{bheight}\ t \uparrow \mathit{bheight}\ u)) \uparrow (1 + (\mathit{bheight}\ v \uparrow \mathit{bheight}\ w)) \\
&= \quad \{\text{由於 } (k+x)\uparrow(k+y) = k + (x \uparrow y),\ (\uparrow)\ \text{有結合律}\} \\
&\quad 1 + (((\mathit{bheight}\ t \uparrow \mathit{bheight}\ u) \uparrow \mathit{bheight}\ v) \uparrow \mathit{bheight}\ w) \\
&= \quad \{\mathit{bheight}\ \text{之定義}\} \\
&\quad 1 + (\mathit{bheight}\ ((\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v)) \uparrow \mathit{bheight}\ w)
\end{aligned}
$$

□

3.2.3 紅黑樹之性質：平衡

之所以討論黑高度，目的之一當然是要討論平衡。我們可證明函數 *ins k* 維持平衡性：

定理 3.5. 對所有 k 與 t, $\mathit{balanced}\ t \Rightarrow \mathit{balanced}\ (\mathit{ins}\ k\ t)$.

而由於 $\mathit{balanced}\ t \Rightarrow \mathit{balanced}\ (\mathit{blacken}\ t)$, 定理 3.5 蘊含 *insert k* 也維持輸入樹的平衡：$\mathit{balanced}\ t \Rightarrow \mathit{balanced}\ (\mathit{insert}\ k\ t)$.

同樣地，要證明定理 3.5，我們也需要一個關於 *rotate* 的引理：

引理 3.6. 對所有 t 與 u,

$$
\begin{aligned}
&\mathit{balanced}\ t \wedge \mathit{balanced}\ u \wedge \\
&\quad \mathit{bheight}\ t = \mathit{bheight}\ u \Rightarrow \mathit{balanced}\ (\mathit{rotate}\ t\ x\ u)\ .
\end{aligned}
$$

由於這些證明與前面提到的證明類似，我們將它們留給讀者做練習。

習題 3.2 — 證明定理 3.5. 你將用得上引理 3.6與定理 3.2.
習題 3.3 — 證明引理 3.6.

3.2.4 紅黑樹之性質：顏色

最後，我們談談紅黑樹插入之後的顏色。我們也許希望函數 $ins\ k$ 能保持準紅黑性，意即 $semiRB\ t \Rightarrow semiRB\ (ins\ k\ t)$. 但這顯然不成立：我們已經知道 $ins\ k$ 可能在一個路徑上產生連續兩個紅節點。

為描述此時的特殊狀況，我們另外定義一個性質：滿足下列述語的樹被稱作紅外 *(infrared)* 樹 — 取自比紅色還紅一點之意：

$$\begin{aligned} &infrared :: \mathsf{RBTree} \to \mathsf{Bool} \\ &infrared\ (\mathsf{R}\ t\ x\ u) = (color\ t == \mathsf{Blk} \vee color\ u == \mathsf{Blk}) \wedge \\ &\qquad\qquad\qquad\qquad semiRB\ t \wedge semiRB\ u \\ &infrared\ t \qquad\qquad = \mathsf{False}\ . \end{aligned}$$

紅外樹幾乎是一棵根部為紅色的準紅黑樹，但兩個子樹 t 與 u 之中最多有一個可以是紅色！我們將其表示成 $color\ t == \mathsf{Blk} \mid color\ u == \mathsf{Blk}$. 此外，$t$ 與 u 仍必須是準紅黑樹。其他的情形（E 或是 $\mathsf{B}\ \_\ \_\ \_$）是黑色的，都不是紅外樹。

那麼，我們是否能證明：給定 $ins\ k$ 準紅黑樹 t，$ins\ k\ t$ 總是一個紅外樹，意即 $semiRB\ t \Rightarrow infrared\ (ins\ k\ t)$？對歸納證明熟悉的讀者可能立刻覺得事有蹊蹺：有二就有三，證明歸納狀況時，如果歸納假設中的子樹有兩個連續的紅節點，在歸納狀況中可能看到三個連續紅節點。如此一來似乎沒完沒了。

這是一個必須嘗試證明一個更強的性質才能使歸納證明成立的例子。函數 ins 真正滿足的是一個更強的性質：給定準紅黑樹 t, 如果 t 是紅色，$ins\ k\ t$ 則是一棵紅外樹；如果 t 是黑色，$ins\ k\ t$ 也將是一棵準紅黑樹：

定理 3.7. *對所有 t:*

1. $semiRB\ t \wedge color\ t = \mathsf{Red} \Rightarrow infrared\ (ins\ k\ t)$,
2. $semiRB\ t \wedge color\ t = \mathsf{Blk} \Rightarrow semiRB\ (ins\ k\ t)$.

為證明定理 3.7，我們也需要一個與 $rotate$ 相關的引理：只要 t 與 u 之中有一個是準紅黑樹，另一個是準紅黑樹或紅外樹，$rotate\ t\ x\ u$ 就會是準紅黑樹：

引理 3.8. *對所有 t 與 u,*

1. $(infrared\ t \vee semiRB\ t) \wedge semiRB\ u \Rightarrow semiRB\ (rotate\ t\ x\ u)$;
2. $semiRB\ t \wedge (infrared\ u \vee semiRB\ u) \Rightarrow semiRB\ (rotate\ t\ x\ u)$.

有了定理 3.7，由於 $infrared\ t \Rightarrow semiRB\ (blacken\ t)$，我們立刻得知 $insert\ k$ 保持準紅黑性：

系理 3.9. $semiRB\ t \Rightarrow semiRB\ (insert\ k\ t)$.

以下證明定理 3.7:

Proof. 定理 3.7的 1, 2 兩個小性質需在同一個歸納中證明。注意：1 與 2 的合取蘊含了

$$semiRB\ t \Rightarrow (infrared\ (ins\ k\ t) \vee semiRB\ (ins\ k\ t))\ . \tag{3.1}$$

我們將用到此性質。

在 t 之上做歸納。我們只舉出兩個具代表性的例子：

狀況: $t := \mathsf{B}\ t\ x\ u$, $k < x$:

$$\begin{aligned}
&semiRB\ (ins\ k\ (\mathsf{B}\ t\ x\ u)) \\
=\quad &\{\ ins\ 之定義, k < x\ \} \\
&semiRB\ (rotate\ (ins\ k\ t)\ x\ u) \\
\Leftarrow\quad &\{\ 引理\ 3.8\ \} \\
&(infrared\ (ins\ k\ t) \vee semiRB\ (ins\ k\ t)) \wedge semiRB\ u \\
\Leftarrow\quad &\{\ 歸納假設，(3.1)\ \} \\
&semiRB\ t \wedge semiRB\ u \\
=\quad &\{\ semiRB\ 之定義\ \} \\
&semiRB\ (\mathsf{B}\ t\ x\ u)\ .
\end{aligned}$$

狀況: $t := \mathsf{R}\ t\ x\ u$, $k < x$:

$$\begin{aligned}
&infrared\ (ins\ k\ (\mathsf{R}\ t\ x\ u)) \\
=\quad &\{\ ins\ 之定義\ \} \\
&infrared\ (\mathsf{R}\ (ins\ k\ t)\ x\ u) \\
=\quad &\{\ infrared\ 之定義\ \} \\
&(color\ (ins\ k\ t) = \mathsf{Blk} \vee color\ u = \mathsf{Blk}) \wedge semiRB\ (ins\ k\ t) \wedge semiRB\ u \\
=\quad &\{\ 歸納假設\ \} \\
&(color\ (ins\ k\ t) = \mathsf{Blk} \vee color\ u = \mathsf{Blk}) \wedge semiRB\ t \wedge color\ t = \mathsf{Blk} \wedge semiRB\ u \\
\Leftarrow\quad &\{\ 命題邏輯: ((\mathsf{P} \vee \mathsf{Q}) \wedge \mathsf{R}) \Leftarrow (\mathsf{Q} \wedge \mathsf{R})\ \} \\
&color\ t = color\ u = \mathsf{Blk} \wedge semiRB\ t \wedge semiRB\ u \\
=\quad &\{\ semiRB\ 之定義\ \} \\
&semiRB\ (\mathsf{R}\ t\ x\ u)\ .
\end{aligned}$$

□

習題 3.4 — 證明引理 3.8.

CHAPTER 4

關於語意的基本概念

第 1章介紹了函數語言的基礎概念，第 2章談了歸納定義與證明。我們將運用這些知識在第 5章之中推導一些程式、解一些編程問題。但在那之前，我們得暫緩一下，釐清一些和語意相關的概念。

語意 (semantics)一詞由語言學中借用而來。語意學是關於字句、詞語的意思的研究。一段程式的「意思」為何？在電腦技術發展的早期這似乎不是個問題。當時，「程式語言」只是屬於某台特定型號電腦的指令集，指令的意思就如同其操作手冊所說。若有疑義，用電腦執行看看就知道了。隨著技術發展，同一個程式語言可在不同電腦上執行，各廠商、研究機構都可以實作同一語言的編譯器。同時，實用性質的程式語言也發展得更加複雜，一個程式語言中常有不太能憑直覺理解的細微處。我們因而需要有個不依賴特定硬體、特定編譯器，也可討論程式語言的「意思」的方法。不僅可作為各家實作程式語言的依據，也方便大家溝通：每當我們設計新的語言、符號，我們也常需要釐清它的語意是什麼。

描述語意的方式有許多種。本書目前為止其實在不知不覺中混用了兩種語意：指稱語意談一個程式*是什麼*，操作語意談一個程式*做什麼*。以下我們以非常粗略的方式介紹它們。

4.1 指稱語意

指稱語意 (denotational semantics)談一個程式*是什麼*。在我們的討論範圍中，有堅實基礎、有不含糊的定義的東西只有數學物件。因此，指稱語意試圖把程式語言對應到數學物件。最簡單的指稱語意可能是基於集合論的：把型別視為集合，把程式語言中的函數視為集合論中的函數。例如，$\mathbb{N}$ 便是自然數的集

合；$(\mathsf{A}\times\mathsf{B})$ 是 A 與 B 的笛卡兒積：[1]

$$(\mathsf{A}\times\mathsf{B}) = \{(a,b) \mid a\in\mathsf{A}, b\in\mathsf{B}\} \ .$$

但函數又是什麼呢？集合論中，一個型別為 $\mathsf{A}\to\mathsf{B}$ 的函數可視為 $\mathsf{A}\times\mathsf{B}$ 的一個子集；若 $f\,x=y$，該子集中便有 (x,y) 這個元素。例如，$double::\mathbb{N}\to\mathbb{N}$ 可表示成如下的集合：

$$\{(0,0),(1,1),(2,4),(3,6),(4,8)\ldots\} \ .$$

能稱為函數的集合還需滿足兩個額外條件：

- 簡單性 (*simplicity*): 對所有 $x\in\mathsf{A}$, 該集合中僅存在一對唯一的 (x,y)。意即每個輸入只對應到一個輸出。
- 完整性 (*totality*): 對所有 $x\in\mathsf{A}$, 在該集合中都存在某對 (x,y)。意即值域中的每個元素都被涵蓋到。

至於遞迴函數呢？以階層 $fact$ 為例，其定義為：

$$\begin{aligned} &fact::\mathbb{N}\to\mathbb{N}\\ &fact\ \mathbf{0} &&= 1\\ &fact\ (\mathbf{1}_+\ n) &&= (\mathbf{1}_+\ n)\times fact\ n \ . \end{aligned}$$

我們可想成：有一個從集合到集合的函數 $factF$，

$$factF\ X \ = \ \{(\mathbf{0},1)\}\cup\{(\mathbf{1}_+\ n,(\mathbf{1}_+\ n)\times m) \mid (n,m)\in X\} \ .$$

給任何集合 X，$factF$ 傳回這樣的集合：

- 新集合中有 $(\mathbf{0},1)$ — 這對應到 $fact\ \mathbf{0}=1$.
- 對 X 之中的每一個 (n,m), 新集合中有 $(\mathbf{1}_+\ n,(\mathbf{1}_+\ n)\times m)$ — 這對應到 $fact\ (\mathbf{1}_+\ n)=(\mathbf{1}_+\ n)\times fact\ n$.

而函數 $fact$ 的語意就是 $factF$ 唯一的定點 (fixed point). 意即 $fact$ 是唯一滿足 $fact=factF\ fact$ 的集合。關於定點的較完整理論可參照第 2.9 節。確實，$fact$ 可寫成集合如下：

$$fact=\{(0,1),(1,1),(2,2),(3,6),(4,24)\ldots\} \ .$$

若將 $fact$ 餵給 $factF$,

$$\begin{aligned} &factF\ fact\\ &=\{(0,1)\}\cup\{(\mathbf{1}_+\ n,(\mathbf{1}_+\ n)\times m) \mid (n,m)\in\{(0,1),(1,1),(2,2),(3,6)\ldots\}\}\\ &=\{(0,1)\}\cup\{(1,1),(2,2),(3,6),(4,24)\ldots\} \ , \end{aligned}$$

我們又得到了 $fact$. 因此 $fact$ 確實是 $factF$ 的定點。

但，要把「$fact$ 便是 $factF$ 的定點」當作 $fact$ 的定義，我們還得確定：確實只有這麼一個集合滿足 $fact=factF\ fact$。讀者稍加檢查一下，即可發現確實如此 — 例如，把 $fact$ 添一項或刪一項，都會使得 $factF\ fact$ 不等於 $fact$.

[1]在更嚴謹的說法中，我們通常用一個語意函數 $[\![_]\!]$將語法對應到其意義。因此我們會說 $[\![\mathsf{A}\times\mathsf{B}]\!]=[\![\mathsf{A}]\!]\times[\![\mathsf{B}]\!]$. 此處採用較不正式的說法。

並不是所有從集合到集合的函數都有唯一的定點。考慮如下兩個函數：

$$\begin{aligned} g\,0 &= 0 \\ g\,x &= g\,(\mathbf{1}_+\,x)\ , \end{aligned} \qquad \begin{aligned} h\,0 &= 0 \\ h\,x &= (-1)\times h\,x\ . \end{aligned}$$

若以類似 $fact$ 的方式試圖將它們寫成某函數的定點，我們可能寫出如下的 gF 和 hF:

$$\begin{aligned} gF\,X &= \{(0,0)\}\cup\{(n,m)\mid(\mathbf{1}_+\,n,m)\in X\}\ , \\ hF\,X &= \{(n,(-1\times m))\mid(n,m)\in X\}\ . \end{aligned}$$

但 gF 的定點有

$$\begin{aligned} &\{(0,0),(1,1),(2,1),(3,1)...\}\ , \\ &\{(0,0),(1,2),(2,2),(3,2)...\}\ , \\ &\{(0,0),(1,10),(2,10),(3,10)...\}\ ... \end{aligned}$$

等無限多個。而 hF 有一個唯一定點：空集合，但空集合違反了前述的完整性要求。

如同 g 和 f 這樣的函數，在我們目前介紹的簡單指稱語意中是沒有定義的。它們沒有語意，在我們的語言中是不該有的存在。它們剛好也是執行起來不會終止的函數。因此，我們似乎可以猜測「有唯一定點」和「執行會終止」似乎有些關聯。但，如前所述，我們的指稱語意中沒有「執行」的觀念。那是操作語意擅長描述的。

值得一提地，指稱語意採取了一個靜態的世界觀。一個函數單純地是輸入與輸出的對應，沒有「執行」的觀念，沒有執行快慢的考量，也沒有預設「終止」的概念。

函數 $double_1\,x = x + x$ 與 $double_2\,x = 2\times x$ 使用不同的演算法，但它們的語意都是同一個集合，因此被視為同一個函數 — 在語意上我們無法區分 $double_1$ 與 $double_2$。

在指稱語意中，當我們說兩個函數相等，意指它們的語意是同一個集合。例如，$map\ \mathbf{1}_+ \cdot concat = concat \cdot map\ (map\ (\mathbf{1}_+))$, 因為兩者的語意都是下述的集合：

$$\{([],[]),...([[1],[2]],[2,3]),...([[1],[2,3]],[2,3,4]),...\}\ .$$

4.2 操作語意

操作語意 (operational semantics)談一個程式做什麼。在操作語意中，我們通常不談符號的「意思」是什麼。符號 $\mathbf{0}$ 只代表它本身，$\mathbf{1}_+\ \mathbf{0}$ 也只代表它本身。操作語意著重的是一串符號如何變成另一段符號。在這個意義下，前述的 $fact$ 定義：

$$\begin{aligned} &fact :: \mathbb{N}\to\mathbb{N} \\ &fact\ \mathbf{0} &&= 1 \\ &fact\ (\mathbf{1}_+\,n) &&= (\mathbf{1}_+\,n)\times fact\ n\ , \end{aligned}$$

從操作語意的觀點可被讀解成覆寫規則：看到 $fact\ \mathbf{0}$, 均可改寫為 1; 看到 $fact\ (\mathbf{1}_+\ n)$, 均可改寫為 $(\mathbf{1}_+\ n) \times fact\ n$. 歸約 $fact\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0}))$ 這個式子可被視為是不斷使用這兩條覆寫規則：

$$\begin{aligned} & fact\ (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})) \\ = & (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})) \times fact\ (\mathbf{1}_+\ \mathbf{0}) \\ = & (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})) \times (\mathbf{1}_+\ \mathbf{0}) \times fact\ \mathbf{0} \\ = & (\mathbf{1}_+\ (\mathbf{1}_+\ \mathbf{0})) \times (\mathbf{1}_+\ \mathbf{0}) \times 1\ , \end{aligned}$$

其中的每個等號都可讀解為「改寫成」。

操作語意中較容易談「執行」與「終止」的概念。改寫一個式子相當於執行它，如果已經沒有可改寫之處，也就是碰到了範式，執行便終止了。在操作語意中，兩個數值（例如 $4+4$ 與 2×4）相等意謂它們可被歸約成同一個範式；兩個函數 f 和 g 相等則意謂對於任何 $x, f\ x$ 與 $g\ x$ 都相等。

有了兩種談語意的方式，我們自然希望它們有些一致性。確實，我們有如下的定理：

定理 4.1. 操作語意保持指稱語意。意即，給定一個有指稱語意的算式 e, 若算式 e 能在零步或多步之內依據操作語意改寫成 e', 則 e 與 e' 的指稱語意相同。

定理 4.2. 給定一個遞迴的函數定義 $f = \mathsf{F}\ f$。若在操作語意中，該函數對所有輸入都正常終止，則在指稱語意中，f 是 F 的唯一定點。

我們在下一節會

CHAPTER 5

一般程式推導

程式推導 (*program derivation*)是本書的重要主題。給定一個待解決的問題，並假設該問題能描述成邏輯、數學、或其他形式語言（稱作一個形式規格(specification)），程式推導泛指以嚴謹方式將該規格轉換成一個解決該問題的程式的方法。指令式語言與函數語言都能做程式推導，而且有許多道理可相通。本書討論函數語言的程式推導。一個簡單、典型的函數程式語言推導可能有如下的形式：

$$
\begin{array}{ll}
& spec \\
= & \quad \{\text{性質 1}\} \\
& e_1 \\
= & \quad \{\text{性質 2}\} \\
& \ldots. \\
= & \quad \{\text{性質 n}\} \\
& e_n \ .
\end{array}
$$

其中 $spec$ 是問題的規格，通常也是一個函數語言程式。根據性質 1, $spec$ 等於 e_1；根據性質 2，e_1 等於 e_2... 長此以往，直到我們導出一個符合我們要求的 e_n. 光這麼看來，這和我們前幾章作的數學證明似乎沒什麼不同。差別在於，做證明時我們已有 $spec$ 和 e_n，要做的是把 $spec$ 到 e_n 之間的步驟補足。但做程式推導時，我們是從 $spec$ 出發，希望透過種種跡象找出一個滿意的（可能是夠快的、不佔空間的、或滿足某些其他性質的）e_n. 當然，找出 e_n 之後，從 $spec$ 推出 e_n 的過程就成了 $spec = e_n$ 的證明。當程式推導的進行方式類似數學演算，有人可能稱之為程式演算 (*program calculation*)。本書混用這兩個詞彙，並不區分。

為何做程式推導？第一個理由是我們希望程式正確。此處「正確」指的是 e_n 確實滿足了最初的規格 $spec$。但如果只為了正確性，我們為何不能先無論如

何把 e_n 寫出，再試著證明 $spec = e_n$？原因之一是通常程式寫好了，大家便不想證它了。更重要的是：程式開發時，若沒有把「怎麼證明它」列入考量，寫出的程式常常是很難證明的。為了確保有證明，最好讓「產生證明」這件事成為程式開發過程的一部分，甚至讓證明引導程式的開發。Dijkstra 說道：

> 先給一個程式再證明它，在某個意義上是把馬車放在馬前面。更有希望的做法是讓正確性證明與程式一起長出來：如此一來，我們能選擇證明的結構，然後設計一個能用這種方法證明出來的程式。這還有個額外的好處：正確性考量可以成為程式該怎麼寫的引導與啟發。*[Dijkstra, 1974]*

Dijkstra 的最後一句話帶到了程式推導的第三個理由：提倡者認為，程式推導的種種方法與技巧可以引導我們去分析、思考、解決問題；這是解問題、甚至發現新演算法的方法（Backhouse [2003], Backhouse [2011]）。藉由良好的符號設計，我們可以發現常見的程式推導模式，並將其推廣到其他有類似性質的問題上。

第 0章的結尾說道「函數語言的價值便是：它是個適於演算的語言。」便於做程式推導，是我覺得函數語言最突出的特質。我們可以由規格出發、在符號上操作，將程式如同求代數的解一樣地算出來。只要每個小步驟都正確，最後的程式就是正確的。由第 1章至今，我們做了許多準備工作，備齊需要的基本知識。現在我們終於可以開始做一些演算了。

5.1 展開-收回轉換

考慮這麼一個例子：給定一個整數形成的串列，我們想計算其每個數的平方的和。例如當輸入是 $[2,6,5,3]$, 我們希望算出 $2^2+6^2+5^2+3^2 = 74$. 這項工作可以簡短地描述如下：

$$sumsq = sum \cdot map\ square \quad . \tag{5.1}$$

函數 *sumsq* 的型別為 List Int → Int. 其中 *map square* 將輸入串列的每個元素都平方，然後由 *sum* 計算其總和。第 2.4節中給過一個 *sum* 的歸納定義，重複如下：

$$\begin{aligned} &sum :: \mathsf{List\ Int} \to \mathsf{Int} \\ &sum\ [\,] \quad\ = 0 \\ &sum\ (x : xs) = x + sum\ xs \quad . \end{aligned}$$

消除中間串列 對大部分的應用而言，如上定義的 *sumsq* 已經很堪用了。但作為一個例子，我們來挑惕些仍不滿意之處。執行 *sumsq xs* 時，*map square* 會產生另一個（存放 *xs* 每個元素的平方的）串列，該串列隨即由 *sum* 消掉 — 感覺上似乎很浪費空間與時間。[1] 有不產生這個中間串列的方法嗎？

[1] 在惰性求值的情況下，該中間串列的每個節點被產生後立刻被 *sum* 消去，因此不會真的佔用和 *xs* 同樣長度的空間。這也是大家覺得惰性求值有助於模組化、使小函數變得易於重用的例子之一。但「產生一個新節點、立刻消去」仍耗了一些不必要的時間。

我們試著做些計算。當 *sumsq* 的參數是 [] 時：

```
  sumsq []
=   { sumsq 之定義 }
  sum (map square [])
=   { map 之定義 }
  sum []
=   { sum 之定義 }
  0 .
```

由此我們得知 *sumsq* [] 會被計算成 0. 當輸入不是空串列時呢？試計算：

```
  sumsq (x:xs)
=   { sumsq 之定義 }
  sum (map square (x:xs))
=   { map 之定義 }
  sum (square x:map square xs)
=   { sum 之定義 }
  square x+sum (map square xs)
=   { sumsq 之定義 }
  square x+sumsq xs  .
```

可得知對任何 x 與 xs, $sum\ (x:xs)$ 歸約的結果和 $square\ x + sumsq\ xs$ 會是相同的。總結說來，藉由計算，我們發現 *sumsq* 滿足以下兩條性質：

$$\begin{aligned} sumsq\ [\,] &= 0 \\ sumsq\ (x:xs) &= square\ x + sumsq\ xs\ \ . \end{aligned} \tag{5.2}$$

但如果我們翻轉過來，把這兩條性質當作 *sumsq* 的新定義呢？這是一個依據歸納法定義的良好程式，而且不會產生中間串列！

但我們怎知道 *sumsq* 的新定義滿足我們最初的要求 (5.1)，即 $sumsq = sum \cdot map\ square$ 呢？回顧起來，我們的演算只證明了當 *sumsq* 滿足 (5.1)，它也滿足 (5.2)，卻還不能據此宣稱另一個方向：若 (5.2) 成立，(5.1) 也成立。我們將在第4章詳細討論這個問題。目前可暫時這麼說：如果我們從某個問題規格 $f = e$ 起始，發現 f 滿足某一組等式，而這些等式剛好可湊成一個會正常終止的歸納定義，則 e 確實是唯一滿足這些等式的解。由於如此的 e 是唯一的，我們也可倒過來以這組等式為 f 的定義，並同時宣稱 $f = e$ 這個性質成立。

回顧起來，在 *sumsq* 的計算中，我們僅是把 *sumsq* 的定義展開，接著展開 *map*, *sum* 等等元件的定義，直到我們又看到 *sumsq* 的定義出現在式子中、剛好可以收回為止。這是一種單純而歷史悠久的程式推導方法，稱作展開-收回轉換 (*fold-unfold transformation*)。雖然簡單，有時這個方法意外地有用。

在這個例子中，我們真正做到的是將一個單行、使用全麥編程的 *sumsq* 定義轉換成了一個歸納定義。新定義的 *sumsq* 比起原版稍有效率些，但這只是恰巧發生的 — 歸納定義的程式不見得總會比較有效率。程式推導確保程式的正確性 — 意即導出的程式與原本的規格是同一個函數。但新程式的效率仍須單獨分析。這是下一節的主題。

習題 5.1 — 下述函數 $descend\ n$ 傳回 $[n, n-1, n-2 \ldots 0]$:

$$\begin{aligned} &descend :: \mathbb{N} \to \mathsf{List}\ \mathbb{N} \\ &descend\ \mathbf{0} \qquad = [\,] \\ &descend\ (\mathbf{1}_+\ n) = \mathbf{1}_+\ n : descend\ n \quad . \end{aligned}$$

定義 $sumseries = sum \cdot descend$. 請找出 $sumseries$ 的歸納定義。

習題 5.2 — 承接習題 5.1。函數 $repeatN :: (\mathbb{N} \times a) \to \mathsf{List}\ a$ 的定義為

$$repeatN\ (n, x) = map\ (const\ x)\ (descend\ n) \quad .$$

因此，$repeatN\ (n, x)$ 會傳回一個含 n 個 x 的串列。例如 $repeatN\ (3, \texttt{'a'}) = \texttt{"aaa"}$. 請算出一個歸納定義的 $repeatN$.

習題 5.3 — 承接習題 5.2。遊程編碼 (run-length encoding)是一種簡單的壓縮方式：將字串中重複的字元表達成其出現的數字。例如 `"aaabbbbcdd"` 可以表達為 `"3a4b1c2d"`. 下列函數 $rld :: \mathsf{List}\ (\mathbb{N} \times a) \to \mathsf{List}\ a$ 則是抽象過的「遊程解碼」，將已經表示成（次數 $\times$字元）的壓縮文展開：

$$rld = concat \cdot map\ repeatN \quad .$$

例如，$rld\ [(2, \texttt{'a'}), (3, \texttt{'b'}), (1, \texttt{'c'})] = \texttt{"aabbbc"}$. 請導出 rld 的歸納定義。

習題 5.4 — 下列函數 $delete$ 將輸入串列中的每個元素輪流刪除：

$$\begin{aligned} &delete\ \ :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &delete\ [\,] \quad = [\,] \\ &delete\ (x : xs) = xs : map\ (x:)\ (delete\ xs) \quad , \end{aligned}$$

例如，$delete\ [1,2,3,4] = [[2,3,4],[1,3,4],[1,2,4],[1,2,3]]$. 函數 $select :: \mathsf{List}\ a \to \mathsf{List}\ (a \times \mathsf{List}\ a)$ 則將一個串列中的元素依次選出。例如，$select\ [1,2,3,4] = [(1,[2,3,4]),(2,[1,3,4]),(3,[1,2,4]),(4,[1,2,3])]$. 函數 $select$ 恰巧可用 $delete$ 定義出來：

$$select\ xs = zip\ xs\ (delete\ xs) \quad .$$

請推導出 $select$ 的歸納定義。**提示**：下述性質可能有用 — $zip\ xs\ (map\ f\ ys) = map\ (id \times f)\ (zip\ xs\ ys)$.

習題 5.5 — 函數 *delete* 有另一個可能定義：

$$\begin{aligned} &delete\ xs = map\ (del\ xs)\ [0\,..\,length\ xs - 1] \\ &\quad \textbf{where}\ del\ xs\ i = take\ i\ xs +\!\!+ drop\ (1+i)\ xs\ \ , \end{aligned}$$

（此處我們利用了當 n 為負數時，$[0\,..\,n]$ 化簡成 $[\,]$ 的特性。）請用此定義推導出 *delete* 的歸納定義。**提示**：你可能用得上下述性質：

$$[0\,..\,n] = 0 : map\ (1+)\ [0\,..\,n-1], \quad \text{if } n \geqslant 0, \tag{5.3}$$

5.2 關於執行效率

有點意外地，本書直到現在才較正式地談執行效率。在本書中，我們假設數字四則運算、邏輯運算元等等都可在常數時間內完成。資料結構方面，使用資料建構元或對其做樣式配對都只需要常數時間。例如，將 x 與 y 做成 (x,y) 是常數時間內可完成的動作；一個型別是序對的值如果已經是弱首範式，將其配對成 (x,y) 以取出其中的 x 和 y 也只需要常數時間。如果有資料結構定義 **data** T = A | B, 給一個已經是弱首範式的 $x ::$ T, 只需常數時間便可判斷它究竟是 A 還是 B.

回顧 List 的定義：

$$\textbf{data}\ \mathsf{List}\ a = [\,] \mid a : \mathsf{List}\ a\ \ .$$

我們可看出這是一個偏一邊的表示法。對於串列的左邊的操作都可以在常數時間內完成。因此，我們可在常數時間內判斷一個串列 xs 究竟是 $[\,]$ 還是可分解成頭和尾；產生 $[\,]$ 只需要常數時間；在 xs 的左邊添加一個元素，傳回 $x : xs$，也是常數時間內可完成的動作。但如果我們要拿出某串列最右邊的元素，或著在串列的右邊加東西呢？回顧 $(+\!\!+)$ 的定義：

$$\begin{aligned} &(+\!\!+) :: \mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a \\ &[\,] \quad\ +\!\!+ ys = ys \\ &(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)\ \ . \end{aligned}$$

若我們試著看看 $[1,2,3] +\!\!+ [4,5]$ 是怎麼被算出來的

$$\begin{aligned} &(1:2:3:[\,]) +\!\!+ (4:5:[\,]) \\ &= 1 : ((2:3:[\,]) +\!\!+ (4:5:[\,])) \\ &= 1 : 2 : ((3:[\,]) +\!\!+ (4:5:[\,])) \\ &= 1 : 2 : 3 : ([\,] +\!\!+ (4:5:[\,])) \\ &= 1 : 2 : 3 : 4 : 5 : [\,]\ \ , \end{aligned}$$

可發現 $(+\!\!+)$ 需把第一個參數從頭到尾走過一遍。因此，若第一個參數的長度是 n，$(+\!\!+)$ 是一個時間複雜度為 $O(n)$ 的函數。函數 *last* 的情況也類似。

習題 5.6 — 回顧 *last* 的定義，試著展開 $last\ [1,2,3,4]$ 並確認 $last\ xs$ 需要的時間是否為 $O(length\ xs)$。

諸如 *sum*, *length*, *maximum* 之類的函數將串列從頭到尾走過一次。當輸入串列長度為 n, 它們均需時 $O(n)$。如果函數 f 需時 $O(t)$，$map\ f$ 需時 $O(t \times n)$. 函數 $filter\ p$, $takeWhile\ p$ 等等在最壞情況下需將串列走完，因此它們也是需要線性時間的函數。函數 *zip* 需要的時間與兩個串列中較短者成正比。

在習題 2.5 中，我們曾請讀者定義一個函數 $reverse :: \mathsf{List}\ a \to \mathsf{List}\ a$，將輸入的串列反轉，例如 $reverse\ [1,2,3,4,5] = [5,4,3,2,1]$. 一個可能的答案如下：

$$\begin{aligned} &reverse\ [\,] &&= [\,] \\ &reverse\ (x : xs) &&= reverse\ xs +\!\!+ [x]\ . \end{aligned}$$

這個程式的效率如何呢？我們看看 $reverse\ [1,2,3,4]$ 如何被歸約：

$$\begin{aligned} &\ \ reverse\ [1,2,3,4] \\ &= reverse\ [2,3,4] +\!\!+ [1] \\ &= (reverse\ [3,4] +\!\!+ [2]) +\!\!+ [1] \\ &= ((reverse\ [4] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\ &= (((reverse\ [\,] +\!\!+ [4]) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\ &= ((([\,] +\!\!+ [4]) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1]\ . \end{aligned}$$

為了把 $[1]$ 接在左邊，$(+\!\!+[1])$ 需要走過一個長度為 3 的串列。而在那之前，$(+\!\!+[2])$ 需要走過一個長度為 2 的串列。推廣說來，要反轉一個長度為 n 的串列，$(+\!\!+)$ 會被使用 $O(n)$ 次。每個 $(+\!\!+)$ 左邊的串列長度也是 $O(n)$，因此 *reverse* 是一個需時 $O(n^2)$ 的演算法！「反轉串列」這個看來很基本的操作竟需要 $O(n^2)$ 的時間，似乎令人難以接受。是否有更快的做法呢？我們將在第 5.6 節討論到。

習題 5.7 — 回顧第 1.10與 2.8節中提及的外標籤二元樹：

$$\mathbf{data}\ \mathsf{ETree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a)\ .$$

以下函數傳回樹中所有的標籤：

$$\begin{aligned} &tips :: \mathsf{ETree}\ a \to \mathsf{List}\ a \\ &tips\ (\mathsf{Tip}\ x) &&= [x] \\ &tips\ (\mathsf{Bin}\ t\ u) &&= tips\ t +\!\!+ tips\ u\ . \end{aligned}$$

函數 *tips* 最壞情況的時間複雜度為何？請做出一個含有 n 個標籤的樹 t，使得 $tips\ t$ 僅需要 $O(n)$ 的時間算完；也請做出一個含有 n 個標籤的樹 u，使得 $tips\ u$ 需要 $O(n^2)$ 的時間。

5.3 用展開-收回轉換增進效率

在前面的例子中，我們手動推導出的 *sumsq* 只比原來的版本快了一點點，並沒有複雜度上的改進。本節我們來看一些使用程式推導改進複雜度的例子。

5.3.1 計算多項式 – Horner 法則

給定整數串列 $as = [a_0, a_1, a_2 \ldots a_n]$ 以及 $x :: \mathsf{Int}$, 我們想計算如下的多項式：

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \ .$$

這問題的規格能清楚寫成：

$$poly\ x\ as = sum\ (zipWith\ (\times)\ as\ (iterate\ (\times x)\ 1)) \ ,$$

其中 $iterate\ (\times x)\ 1$ 產生無限串列 $[1, x, x^2, x^3 \ldots]$, $zipWith$ 計算 $[a_0, a_1 x, \ldots a_n x^n]$, sum 計算總和。

讀者應已對 *sum* 和 *zipWith* 很熟悉了。函數 *iterate* 在第 1.8.2 節中使用過，$iterate\ f\ x$ 會展開為無限長的串列 $[x, f\ x, f\ (f\ x) \ldots]$，每個元素分別是把 f 使用 0次、1次、2次... 的結果。函數 *iterate* 可定義為

$$\begin{aligned} &iterate :: (a \to a) \to a \to \mathsf{List}\ a \\ &iterate\ f\ x = x : map\ f\ (iterate\ f\ x) \ . \end{aligned}$$

我們可將之理解成：$iterate\ f\ x$ 的第一個元素是 x；剩下的元素呢？是把 $iterate\ f\ x$ 本身拿來，對每個元素多做一次 f!

讀到此的讀者可能有些疑問：這是一個合法的歸納定義嗎？以及，我們原已說定不談無限的資料結構，何以在此卻出現了呢？

上述 *iterate* 的定義方式確實不是歸納，而是「餘歸納」(coinduction) 的一個例子。第 2.9節中曾提及，餘歸納與歸納互為對偶，以餘歸納定義出的資料結構稱作「餘資料」，可以是能無限地展開的。「餘串列」和歸納定義的串列應該視為不同的型別，但它們可共存於同一個程式中，只要我們確定不在餘資料上做歸納定義或證明。

為培養一些對 *iterate* 的直覺，我們試著展開它：

$$\begin{aligned} & iterate\ f\ x \\ = \quad & \{\ iterate\ 之定義\ \} \\ & x : map\ f\ (iterate\ f\ x) \\ = \quad & \{\ map\ 之定義\ \} \\ & x : map\ f\ (x : map\ f\ (iterate\ f\ x)) \\ = \quad & \{\ map\ 融合\ \} \\ & x : f\ x : map\ (f \cdot f)\ (iterate\ f\ x) \\ = \quad & \{\ iterate\ 與\ map\ 之定義\ \} \\ & x : f\ x : f\ (f\ x) : map\ (f \cdot f)\ (map\ f\ (iterate\ f\ x)) \\ = \quad & \{\ map\ 融合\ \} \\ & x : f\ x : f\ (f\ x) : map\ (f \cdot f \cdot f)\ (iterate\ f\ x) \end{aligned}$$

可發現越展開，式子中便累積越多個 $map\,f$.

在 $poly$ 之中，$iterate$ 雖產生無限長的餘資料，但立刻被 $zipWith$ 截短了。[2] 若我們試著展開 $poly\,x\,[a,b,c,d]$, 會得到：

$$\begin{aligned}
&poly\,x\,[a,b,c,d]\\
=\;&sum\,(zipWith\,(\times)\,[a,b,c,d]\,(iterate\,(\times x)\,1))\\
=\;&\quad \{\text{展開 } iterate\text{, 將「}f\text{自我組合四次」記為 } f^4\,\}\\
&sum\,(zipWith\,(\times)\,[a,b,c,d]\\
&\quad (1:(1\times x):(1\times x\times x):(1\times x\times x\times x):map\,(\times x)^4\,(iterate\,(\times x)\,1)))\\
=\;&a+b\times x+c\times x\times x+d\times x\times x\times x\;\;.
\end{aligned}$$

可看到式子越長，便累積越多個 $(\times x)$。當 as 長度為 n, 需要的乘法數目為 $O(n^2)$. 我們有可能降低做乘法的次數嗎？

我們試著找出 $poly$ 在 as 上的歸納定義。當 $as := [\,]$ 時，$poly\,x\,[\,]$ 可歸約為 0. 考慮 $as := a:as$ 的情況，和做證明時一樣，我們先將 $poly\,x\,(a:as)$ 展開，然後試著整理出 $sum\,(zipWith\,(\times)\,as\,(iterate\,(\times x)\,1)$，以便收回成為 $poly\,x\,as$。計算中的每一步都以此為目的，試著將 sum 與 $zipWith$ 移動至 $iterate$ 旁邊:

$$\begin{aligned}
&poly\,x\,(a:as)\\
=\;&\quad \{poly\text{ 的定義}\}\\
&sum\,(zipWith\,(\times)\,(a:as)\,(iterate\,(\times x)\,1))\\
=\;&\quad \{iterate\text{ 的定義}\}\\
&sum\,(zipWith\,(\times)\,(a:as)\,(1:map\,(\times x)\,(iterate\,(\times x)\,1)))\\
=\;&\quad \{zipWith\text{ 與 }sum\text{ 的定義}\}\\
&a+sum\,(zipWith\,(\times)\,as\,(map\,(\times x)\,(iterate\,(\times x)\,1)))\\
=\;&\quad \{zipWith\,(\times)\,as\cdot map\,(\times x)=map\,(\times x)\cdot zipWith\,(\times)\,as\text{，見習題}\}\\
&a+sum\,(map\,(\times x)\,(zipWith\,(\times)\,as\,(iterate\,(\times x)\,1)))\\
=\;&\quad \{sum\cdot map\,(\times x)=(\times x)\cdot sum\}\\
&a+(sum\,(zipWith\,(\times)\,as\,(iterate\,(\times x)\,1)))\times x\\
=\;&\quad \{poly\text{ 的定義}\}\\
&a+(poly\,x\,as)\times x\;\;.
\end{aligned}$$

第 4 步中關於 $zipWith$ 與 map 的性質幫助我們將 $map\,(\times x)$ 往外提、將 $zipWith$ 往裡推。事實上，該性質不限於乘法，而可適用於任何滿足結合律的運算子 $(\otimes)$。我們可非正式地理解如下：

$$\begin{aligned}
&zipWith\,(\otimes)\,[a,b,c]\,(map\,(\otimes x)\,[d,e,f])\\
=\;&[a\otimes(d\otimes x),b\otimes(e\otimes x),c\otimes(f\otimes x)]\\
=\;&\quad \{\text{結合律: } m\otimes(n\otimes k)=(m\otimes n)\otimes k\}\\
&[(a\otimes d)\otimes x,(b\otimes e)\otimes x,(c\otimes f)\otimes x]\\
=\;&map\,(\otimes x)\,(zipWith\,(\otimes)\,[a,b,c]\,[d,e,f])\;\;.
\end{aligned}$$

第 5 步之中的 $sum\cdot map\,(\times x)=(\times x)\cdot sum$ 在習題 2.7中證明過，需要乘法與加法的分配律。在本推導中，它的功能是將 sum 往右推。它也是使 $poly$ 可以加

[2] 此處我們假設 as 為有限長的串列，並把 $zipWith$ 視為在其第一個參數之上的歸納定義。

速的關鍵性質：共同的 $(\times x)$ 可以提出來 — 左手邊可能做了的許多次 $(\times x)$ 其實只需做一次。追根究底，*poly* 之所以能算得更快，都歸功於乘法與加法的分配律。經過上述計算，我們可得：

$$\begin{aligned} &poly\ x\ [\,] &&= 0 \\ &poly\ x\ (a : as) &&= a + (poly\ as) \times x\ . \end{aligned}$$

在這個定義中，函數 *poly* 遞迴多少次，便做多少個乘法。因此本演算法所需的乘法數目為 $O(n)$.

快速版本的函數 *poly* 相當於把 $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ 轉換成

$$a_0 + x \times (a_1 + x \times (a_2 + \ldots + (a_{n-1} + x \times a_n)))\ .$$

這條規則在 William George Horner 1819 年的一篇論文中出現並證明，因此通常被稱作 *Horner* 法則，雖然 Horner 本人和許多歷史學家們都相信該規則可被追溯得更早。

習題 5.8 — 試證明：如果 $(\otimes)$ 滿足結合律，$zipWith\ (\otimes)\ as \cdot map\ (\otimes x) = map\ (\otimes x) \cdot zipWith\ (\otimes)\ as$。

5.3.2 二進位表示法

回顧第 2.2節中的函數 *exp*。該函數計算乘冪 — $exp\ b\ n = b^n$，其定義如下:

$$\begin{aligned} &exp :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &exp\ b\ \mathbf{0} &&= 1 \\ &exp\ b\ (\mathbf{1}_+\ n) &&= b \times exp\ b\ n\ . \end{aligned}$$

以這個演算法計算 b^n 時會需要 $O(n)$ 個乘法。是否有更快的做法呢？

我們先定義函數 $binary :: \mathbb{N} \to \text{List Bool}$，可將一個自然數轉換成反轉的二進位表示法（即最低有效位在左邊，最高有效位在右邊）。以下我們將 False 簡寫為 $\mathbb{0}$, True 簡寫為 $\mathbb{1}$：

$$\begin{aligned} &binary\ 0 = [\,] \\ &binary\ n \mid even\ n &&= \mathbb{0} : binary\ (n\ `div`\ 2) \\ &\qquad\quad \mid odd\ n &&= \mathbb{1} : binary\ (n\ `div`\ 2)\ . \end{aligned}$$

例如，$map\ binary\ [1,2,3,4] = [[\mathbb{1}],[\mathbb{0},\mathbb{1}],[\mathbb{1},\mathbb{1}],[\mathbb{0},\mathbb{0},\mathbb{1}]]$. 函數 *binary* 在每次遞迴呼叫時將參數減半，因此 $binary\ n$ 只需要 $O(\log n)$ 的時間。我們讓 *binary* 傳回反轉的二進位數是為了方便定義 $decimal :: \text{List Bool} \to \mathbb{N}$ — *binary* 的反函數，將 *binary* 的結果轉回成原有的數字：

$$\begin{aligned} &decimal\ [\,] &&= 0 \\ &decimal\ (c : cs) &&= \mathbf{if}\ c\ \mathbf{then}\ 1 + 2 \times decimal\ cs\ \mathbf{else}\ 2 \times decimal\ cs\ . \end{aligned}$$

我們可證明 $decimal \cdot binary = id$.

回到 *exp*, 試計算如下

$$
\begin{aligned}
& exp\ b \\
= & \quad \{\ id \text{ 為 } (\cdot) \text{ 的單位元}\ \} \\
& exp\ b \cdot id \\
= & \quad \{\ decimal \cdot binary = id\ \} \\
& exp\ b \cdot decimal \cdot binary\ .
\end{aligned}
$$

由於 $binary\ n$ 只需 $O(\log n)$ 的時間，如果我們能把 $exp\ b \cdot decimal$ 的計算時間也縮減到 $O(\log n)$, 我們就有個只需對數時間的演算法了！

令 $roll\ b = exp\ b \cdot decimal$。顯然 $roll\ b\ [\,] = 1$. 考慮輸入為 $c : cs$ 的情況，在以下的推導中，我們假設 $exp\ b\ n = b^n$ 擁有乘冪該有的各種算術性質：

$$
\begin{aligned}
& roll\ b\ (c : cs) \\
= & \quad \{\ exp2 \text{ 與 } decimal \text{ 之定義}\ \} \\
& exp\ b\ (\textbf{if}\ c\ \textbf{then}\ 1 + 2 \times decimal\ cs\ \textbf{else}\ 2 \times decimal\ cs) \\
= & \quad \{\ \text{函數分配進 } \textbf{if}\ \} \\
& \textbf{if}\ c\ \textbf{then}\ exp\ b\ (1 + 2 \times decimal\ cs)\ \textbf{else}\ exp\ b\ (2 \times decimal\ cs) \\
= & \quad \{\ \text{算術：} b^{i+2\times x} = b^i \times (b^x)^2\ \} \\
& \textbf{if}\ c\ \textbf{then}\ b \times square\ (exp\ b\ (decimal\ cs))\ \textbf{else}\ square\ (exp\ b\ (decimal\ cs)) \\
= & \quad \{\ roll \text{ 之定義}\ \} \\
& \textbf{if}\ c\ \textbf{then}\ b \times square\ (exp2\ b\ cs)\ \textbf{else}\ square\ (roll\ b\ cs)\ .
\end{aligned}
$$

因此，我們推導出了在 $O(\log n)$ 時間內計算 b^n 的程式如下：

$$
\begin{aligned}
& exp\ b = roll\ b \cdot binary \\
& roll\ b\ [\,] \quad\ = 1 \\
& roll\ b\ (c : cs) = \textbf{if}\ c\ \textbf{then}\ b \times square\ (exp2\ b\ cs)\ \textbf{else}\ square\ (exp2\ b\ cs)\ .
\end{aligned}
$$

習題 5.9 — 如何得知 $binary$ 會終止？它的定義用的是什麼歸納方式？

習題 5.10 — 證明 $decimal \cdot binary = id$.

習題 5.11 — 展開 $exp\ b = roll\ b \cdot binary$ 以導出一個不產生中間串列的 exp 定義。這個定義用的是什麼歸納方式呢？

5.3.3 小結與提醒

如果回顧本節發生了什麼，該說：我們為 $poly$ 和 $roll \cdot decimal$ 找出了歸納定義。它們的效率因此提升了，但這只能說是湊巧：兩個演算中，都有些代數性質可運用，使得推導出的歸納定義在每一步需要做的工作不多，剛好是有效率的。

一般說來，歸納定義和效率提升不見得能畫上等號。導出了某函數的歸納定義後，仍需針對它做分析，才能知道這個推導是否值得。有些情況下，推導出的程式有較好的時間複雜度，這樣的程式通常能表現得比原來的程式好。有些情況下，找出歸納定義能消除中間串列，或著減少走訪資料結構的次數。這時，導出的程式仍有同樣的時間複雜度，但可能有較小的常數。此時需注意：

本章談的僅是時間複雜度，而且只考慮一些特定運算元被使用的次數。實際上的執行效率受到許多因素的影響，例如：不同運算元花費不同的時間；空間的使用量也影響效率（例如，使用大量記憶體的程式可能需要較多次垃圾收集）；某些演算法適合快取，等等。我們在之後的章節中將看到一些歸納定義程式走訪資料結構的次數雖較少，但反而執行得慢的例子。

5.4 變數換常數

接下來的幾節中，我們將介紹幾種常見的程式推導技巧。[todo: generalize]

本節先以一個簡單但時常用上的技巧作為開頭。回顧在例 1.26 中提及的函數 *positions*:

$$positions\ z = map\ fst \cdot filter\ ((==z) \cdot snd) \cdot zip\ [\mathbf{0}..]\ .$$

positions z xs 傳回 *z* 在 *xs* 中出現的所有位置。上述的定義方式會產生許多中間串列。我們能用第 5.1 節中的方式，利用展開-收回轉換為 *positions* 推導出一個歸納定義，並藉此將中間串列消除嗎？

我們針對 *positions z* 的輸入做分析，在歸納狀況中，先將 *positions z* $(x:xs)$ 展開，希望最後能收回 *positions z xs*. 由於讀者對相關計算應已熟悉，以下的演算以較快的步調進行：

$$\begin{aligned}
&positions\ z\ (x:xs)\\
&= map\ fst \cdot filter\ ((==z) \cdot snd) \cdot zip\ [\mathbf{0}..]\ \$\ (x:xs)\\
&= \quad \{ zip\ 之定義，[0..] = 0:[1..] \}\\
&\quad map\ fst \cdot filter\ ((==z) \cdot snd)\ \$\ (\mathbf{0},x) : zip\ [1..]\ (x:xs)\\
&= \quad \{ map\ 與\ filter\ 之定義 \}\\
&\quad \mathbf{if}\ x == z\ \mathbf{then}\ \mathbf{0} : map\ fst\ (filter\ ((==z) \cdot snd)\ (zip\ [1..]\ (x:xs)))\\
&\qquad \mathbf{else}\ map\ fst\ (filter\ ((==z) \cdot snd)\ (zip\ [1..]\ (x:xs)))\ .
\end{aligned}$$

結果我們卡在這兒了：最後的式子中，*zip* 的參數是 $[1..]$, 但 *positions* 的定義要求這個參數得是 $[\mathbf{0}..]$. 我們無法將式子收回得到 *positions z xs*.

看來，問題是 *positions* 的定義把 **0** 給寫死了。如果我們索性把該位置變成一個變數呢？我們定義：

$$posFrom\ z\ i = map\ fst \cdot filter\ ((z==) \cdot snd) \cdot zip\ [i..]$$

函數 *positions* 是 *posFrom* 的特例：$positions\ z = posFrom\ z\ 0$. 而 *posFrom* 的歸納定義可用展開-收回轉換找出：

$$\begin{aligned}
&posFrom\ z\ i\ (x:xs)\\
&= map\ fst \cdot filter\ ((==z) \cdot snd) \cdot zip\ [i..]\ \$\ (x:xs)\\
&= \quad \{ zip\ 之定義，[i..] = i:[\mathbf{1}_+\ i..] \}\\
&\quad map\ fst \cdot filter\ ((==z) \cdot snd)\ \$\ (i,x) : zip\ [\mathbf{1}_+\ i..]\ (x:xs)\\
&= \quad \{ map\ 與\ filter\ 之定義 \}\\
&\quad \mathbf{if}\ x == z\ \mathbf{then}\ i : map\ fst\ (filter\ ((==z) \cdot snd)\ (zip\ [\mathbf{1}_+\ i..]\ (x:xs)))\\
&\qquad \mathbf{else}\ map\ fst\ (filter\ ((==z) \cdot snd)\ (zip\ [\mathbf{1}_+\ i..]\ (x:xs)))
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \mathit{posFrom} \text{ 之定義} \} \\
&\quad \mathbf{if}\ x == z\ \mathbf{then}\ i : \mathit{posFrom}\ z\ (\mathbf{1}_{+}\ i)\ \mathit{xs} \\
&\qquad \mathbf{else}\ \mathit{posFrom}\ z\ (\mathbf{1}_{+}\ i)\ \mathit{xs}\ \ .
\end{aligned}
$$

由此我們可得：

$$
\begin{aligned}
\mathit{posFrom}\ z\ i\ [\,] &= [\,] \\
\mathit{posFrom}\ z\ i\ (x : \mathit{xs}) &= \mathbf{if}\ x == z\ \mathbf{then}\ i : \mathit{posFrom}\ z\ (1+i)\ \mathit{xs} \\
&\qquad\qquad \mathbf{else}\ \mathit{posFrom}\ z\ (1+i)\ \mathit{xs}\ \ .
\end{aligned}
$$

由於將一個常數換成變數，*posFrom* 比 *positions* 多了些彈性，因此較容易收回。這個技巧在許多場合用得上，往往是許多程式推導的第一步。但我並不建議大家看到任一個定義，便一股腦地把所有常數都換成變數。如同第 2.7節中提及的原則，該把哪些常數換掉仍應由計算中發現，只用在必要之處。

習題 5.12 — 函數 *index* 為串列中的每個元素標上位置。

$$
\begin{aligned}
&\mathit{index} :: \mathsf{List}\ a \to \mathsf{List}\ (\mathbb{N} \times a) \\
&\mathit{index} = \mathit{zip}\ [\mathbf{0}\,..\,]\ \ .
\end{aligned}
$$

請試著為 *index* 導出一個歸納定義。如果不成功，找出一個更通用的輔助函數，並導出其歸納定義。

5.5 組對

接下來我們介紹另一個重要的技巧：「組對 (tupling)」。

5.5.1 陡串列

給定一個整數串列。當我們說它很「陡」，意思是它由左到右下降得極快，快到每一個元素都大於其右邊所有元素的和。形式化的定義如下：

$$
\begin{aligned}
&\mathit{steep} :: \mathsf{List\ Int} \to \mathsf{Bool} \\
&\mathit{steep}\ [\,] \quad\ = \mathsf{True} \\
&\mathit{steep}\ (x : \mathit{xs}) = x > \mathit{sum}\ \mathit{xs} \wedge \mathit{steep}\ \mathit{xs}\ \ .
\end{aligned}
$$

如果當作一個程式，當輸入串列長度為 n, 由於反覆呼叫 *sum*, 上述的程式需要 $O(n^2)$ 的時間。但每次算出的 $\mathit{sum}\ \mathit{xs}$ 和 x 比較後便立刻被丟棄，似乎很浪費。我們能否把 *sum* 的結果存下來呢？下述函數 *steepsum* 把 *steep* 與 *sum* 都算出來，放在一個序對中：

$$
\begin{aligned}
&\mathit{steepsum} :: \mathsf{List\ Int} \to (\mathsf{Bool} \times \mathsf{Int}) \\
&\mathit{steepsum}\ \mathit{xs} = (\mathit{steep}\ \mathit{xs}, \mathit{sum}\ \mathit{xs})\ \ .
\end{aligned}
$$

我們看看 *steepsum* 能否算得快一點？

根據定義，$\mathit{steepsum}\ [\,] = (\mathsf{True}, 0)$. 我們看看 $\mathit{xs} = x : \mathit{xs}$ 的例子：

$$
\begin{aligned}
& steepsum\ (x : xs) \\
= & \quad \{\ steepsum\ 之定義\ \} \\
& (steep\ (x : xs), sum\ (x : xs)) \\
= & \quad \{\ steep\ 與\ sum\ 之定義\ \} \\
& (x > sum\ xs \wedge steep\ xs, x + sum\ xs) \\
= & \quad \{\ 將子算式提取至\ \mathbf{let}\ 中\ \} \\
& \mathbf{let}\ (b, s) = (steep\ xs, sum\ xs) \\
& \mathbf{in}\ (x > s \wedge b, x + s) \\
= & \quad \{\ steepsum\ 之定義\ \} \\
& \mathbf{let}\ (b, s) = steepsum\ xs \\
& \mathbf{in}\ (x > s \wedge b, x + s)\ .
\end{aligned}
$$

我們已推導出：

$$
\begin{aligned}
steepsum\ [\,] &= (\mathsf{True}, 0) \\
steepsum\ (x : xs) &= \mathbf{let}\ (b, s) = steepsum\ xs \\
&\quad\ \mathbf{in}\ (x > s \wedge b, x + s)\ .
\end{aligned}
$$

這是一個只用 $O(n)$ 時間的程式。有了 *steepsum*，我們可重新定義 *steep* 為 $steep = fst \cdot steepsum$。

讓一個函數多傳回一些值的動作稱作組對 (*tupling*) — 因為多傳回的值被放在一個序對中。[3] 我們常用此技巧來存下可重複使用的中間值，並減少走訪資料結構的次數。

最後一提：若使用第 1.6.3節，36頁介紹的「分裂」運算元：

$$
\begin{aligned}
& \langle \cdot, \cdot \rangle :: (a \to b) \to (a \to c) \to a \to (b \times c) \\
& \langle f, g \rangle\ x = (f\ x, g\ x)\ .
\end{aligned}
$$

函數 *steepsum* 的定義可較簡潔地寫成：

$$
steepsum = \langle steep, sum \rangle\ .
$$

本書將在適當時採用這種寫法。

責任越大，能力越強？ 某個意義上，函數 *steepsum* 做的事比 *steep* 多：後者只判斷輸入是否為陡串列，前者不只如此，還多附送了串列的和。然而，傳回較多東西、似乎做了更多事的程式，反倒可以執行得比較快。竟出現「責任越大，能力越強」這種違反直覺的現象，這是怎麼回事呢？[4]

其實，在歸納定義與歸納證明中，這都是常見的。有些比較通用的程式反倒比較容易定義；有些定理本身不好證明，為了證明它，我們把它變得更廣泛、更強些，反倒好證了。箇中原因說穿了便不難理解，函數 *steepsum* 便是

[3] 在一些函數語言中，'pair" 指含兩個成員的序對，'tuple" 則不限定為兩個成員。Tuple 也可當作動詞，指做出一個 tuple. 本書將動詞的 tuple 譯為「組對」– 組出一個對。有些語言中 pair 與 tuple 有更根本的差異，但本書中不做區分。

[4] 「能力越強，責任越大 (with great power comes great responsibility)」是 2002 年版《蜘蛛人》電影的名句。根據考證 [O'Toole, 2017], 法國國民公會 1793 年的政令中即出現過類似的想法，包括邱吉爾和羅斯福等人也都說過類似話語的不同版本。

一個容易明白的好例子：一個函數若傳回較多資訊，在遞迴呼叫它時，我們便有更多資訊可直接取用。同樣地，一個定理若保證更強的性質，表示在使用歸納假設的步驟中，我們有了更強的性質可用。

如果我們把一個函數或待證的定理擴充得太強，確實也有可能使它們強到寫不出來、證不出來。做歸納定義或證明的重要技巧之一，便是找到這麼一個平衡點：將一個待定義或證明的物件擴充到足以提供歸納步驟需要的所有資訊，又不至於強到無法寫出、證出。

在這一節以及下一節中，我們都會看到許多如此的例子。

5.5.2 以串列標記樹狀結構

我們再舉一個組對的好例子。回顧第 1.10與 2.8節中提及的外標籤二元樹：

$$\textbf{data}\ \mathsf{ETree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a)\ .$$

下述函數 $size :: \mathsf{ETree}\ a \to \mathsf{Int}$ 計算一棵樹中標籤的數目（對 ETree 而言是 Tip 出現的次數）；$repl\ t\ xs$ 則將 t 原有的標籤丟棄，改用串列 xs 由右至左依序重新為 t 上標籤。觀察：在遞迴呼叫中，我們用 $take$ 和 $drop$ 將串列 xs 截成適當的長度：

$$\begin{aligned}
&size\ (\mathsf{Tip}\ _) &&= 1\\
&size\ (\mathsf{Bin}\ t\ u) &&= size\ t + size\ u\ ,\\
&repl :: \mathsf{ETree}\ a \to \mathsf{List}\ b \to \mathsf{ETree}\ b\\
&repl\ (\mathsf{Tip}\ _)\ \ xs &&= \mathsf{Tip}\ (head\ xs)\\
&repl\ (\mathsf{Bin}\ t\ u)\ xs &&= \mathsf{Bin}\ (repl\ t\ (take\ n\ xs))\ (repl\ u\ (drop\ n\ xs))\\
&\quad \textbf{where}\ n = size\ t\ .
\end{aligned}$$

如果 t 是一個向左傾斜的二元樹，$repl\ t\ xs$ 不僅會反覆計算 $size$，也會反覆地將 $take$ 與 $drop$ 用在 xs 上，使得上述的 $repl$ 成為一個 $O(n^2)$ 的演算法。利用組對的技巧，我們能讓 $repl$ 的時間複雜度小一些嗎？

我們試著把 $repl$ 作用在一個稍微左斜的樹，$(\mathsf{Bin}\ (\mathsf{Bin}\ t\ u)\ v)$ 之上，看看有什麼能做的。令 t, u 的 $size$ 分別為 n_1 與 n_2. 如果我們希望導出一個線性時間的程式，其中一個提示是：我們希望在 $repl\ (\mathsf{Bin}\ (\mathsf{Bin}\ t\ u)\ v)\ xs$ 中，xs 的每個元素最多都只被 $take$ 和 $drop$ 各碰過一次。依此原則，以下的推導基嘗試做到兩點：首先，把連續的 $take$ 消去；其次，若已做了 $take\ n_1\ xs$，就避免再出現 $take\ (n_1 + n_2)\ xs$，因為後者的存在會讓 $take$ 重複處理 xs 中的元素。我們會用到習題 2.38 – 2.40 中提到的三個性質：

$$\begin{aligned}
take\ m\ (take\ (m+n)\ xs) &= take\ m\ xs\ ,\\
drop\ m\ (take\ (m+n)\ xs) &= take\ n\ (drop\ m\ xs)\ ,\\
drop\ (m+n)\ xs &= drop\ n\ (drop\ m\ xs)\ .
\end{aligned}$$

其中 m, n 均為自然數。試演算如下：

$$\begin{aligned}
&repl\ (\mathsf{Bin}\ (\mathsf{Bin}\ t\ u)\ v)\ xs\\
=\ &\quad \{\ repl\ \text{之定義，兩次}\ \}
\end{aligned}$$

$$
\begin{aligned}
&\mathsf{Bin}\ (\mathsf{Bin}\ (\mathit{repl}\ t\ (\mathit{take}\ n_1\ (\mathit{take}\ (n_1+n_2)\ \mathit{xs})))\\
&\qquad\qquad (\mathit{repl}\ u\ (\mathit{drop}\ n_1\ (\mathit{take}\ (n_1+n_2)\ \mathit{xs}))))\\
&\qquad (\mathit{repl}\ v\ (\mathit{drop}\ (n_1+n_2)\ \mathit{xs}))\\
=\quad &\{\ \text{習題 2.38 – 2.40}\ \}\\
&\mathsf{Bin}\ (\mathsf{Bin}\ (\mathit{repl}\ t\ (\mathit{take}\ n_1\ \mathit{xs}))\\
&\qquad\qquad (\mathit{repl}\ u\ (\mathit{take}\ n_2\ (\mathit{drop}\ n_1\ \mathit{xs}))))\\
&\qquad (\mathit{repl}\ v\ (\mathit{drop}\ n_2\ (\mathit{drop}\ n_1\ \mathit{xs})))\ .
\end{aligned}
$$

演算到此，與 t 有關的是 $\mathit{repl}\ t$, $\mathit{take}\ (\mathit{size}\ t)$, 與 $\mathit{drop}\ (\mathit{size}\ t)$ 三項；與 u 有關的是 $\mathit{repl}\ u$, $\mathit{take}\ (\mathit{size}\ u)$, 與 $\mathit{drop}\ (\mathit{size}\ u)$ 三項。如果我們把 $\mathit{repl}\ t$, $\mathit{take}\ (\mathit{size}\ t)$, 與 $\mathit{drop}\ (\mathit{size}\ t)$ 取出，當作一個函數之內完成的動作：

$$
\begin{aligned}
&\mathit{repTail} :: \mathsf{ETree}\ a \to \mathsf{List}\ b \to (\mathsf{ETree}\ b \times \mathsf{List}\ b)\\
&\mathit{repTail}\ s\ \mathit{xs} = (\mathit{repl}\ s\ (\mathit{take}\ n\ \mathit{xs}), \mathit{drop}\ n\ \mathit{xs})\ ,\\
&\quad \textbf{where}\ n = \mathit{size}\ s\ .
\end{aligned}
$$

那麼 $\mathsf{Bin}\ (\mathit{repl}\ t\ (\mathit{take}\ n_1\ \mathit{xs}))\ (\mathit{repl}\ u\ (\mathit{take}\ n_2\ (\mathit{drop}\ n_1\ \mathit{xs})))$ 似乎有可能收回成為這樣的式子：xs 先被丟給 $\mathit{repTail}\ t$，將 t 標記好，並得到剩下的串列 $\mathit{drop}\ n_1\ \mathit{xs}$。這個剩下的串列又可以丟給 $\mathit{repTail}\ u$, 兩者都只把 xs 走過一次。我們試著導出 $\mathit{repTail}$ 的歸納定義。基底狀況 $s := \mathsf{Tip}\ y$ 比較容易，我們考慮 $s := \mathsf{Bin}\ t\ u$ 的情況，並演算如下（令 $n_1 = \mathit{size}\ t$，$n_2 = \mathit{size}\ u$, 因此 $\mathit{size}\ (\mathsf{Bin}\ t\ u) = n_1 + n_2$）：

$$
\begin{aligned}
&\mathit{repTail}\ (\mathsf{Bin}\ t\ u)\ \mathit{xs}\\
=\quad &\{\ \mathit{repTail}\ \text{之定義}\ \}\\
&(\mathit{repl}\ (\mathsf{Bin}\ t\ u)\ (\mathit{take}\ (n_1+n_2)\ \mathit{xs}), \mathit{drop}\ (n_1+n_2)\ \mathit{xs})\\
=\quad &\{\ \mathit{repl}\ \text{之定義，令}\ n_1 = \mathit{size}\ t\ \}\\
&(\mathsf{Bin}\ (\mathit{repl}\ t\ (\mathit{take}\ n_1\ (\mathit{take}\ (n_1+n_2)\ \mathit{xs})))\\
&\qquad (\mathit{repl}\ u\ (\mathit{drop}\ n_1\ (\mathit{take}\ (n_1+n_2)\ \mathit{xs}))), \mathit{drop}\ (n_1+n_2)\ \mathit{xs})\\
=\quad &\{\ \text{習題 2.38 – 2.40}\ \}\\
&(\mathsf{Bin}\ (\mathit{repl}\ t\ (\mathit{take}\ n_1\ \mathit{xs}))\\
&\qquad (\mathit{repl}\ u\ (\mathit{take}\ n_2\ (\mathit{drop}\ n_1\ \mathit{xs}))), \mathit{drop}\ n_2\ (\mathit{drop}\ n_1\ \mathit{xs}))\\
=\quad &\{\ \text{提出共同項}\ \}\\
&\textbf{let}\ (t', \mathit{xs}') = (\mathit{repl}\ t\ (\mathit{take}\ n_1\ \mathit{xs}), \mathit{drop}\ n_1\ \mathit{xs})\\
&\qquad (u', \mathit{xs}'') = (\mathit{repl}\ u\ (\mathit{take}\ n_2\ \mathit{xs}'), \mathit{drop}\ n_2\ \mathit{xs}')\\
&\textbf{in}\ (\mathsf{Bin}\ t'\ u', \mathit{xs}'')\\
=\quad &\{\ \mathit{repTail}\ \text{之定義}\ \}\\
&\textbf{let}\ (t', \mathit{xs}') = \mathit{repTail}\ t\ \mathit{xs}\\
&\qquad (u', \mathit{xs}'') = \mathit{repTail}\ u\ \mathit{xs}'\\
&\textbf{in}\ (\mathsf{Bin}\ t'\ u', \mathit{xs}'')\ .
\end{aligned}
$$

因此我們得到：

$$
\begin{aligned}
&\mathit{repTail}\ (\mathsf{Tip}\ \_)\ \ \mathit{xs} = (\mathsf{Tip}\ (\mathit{head}\ \mathit{xs}), \mathit{tail}\ \mathit{xs})\\
&\mathit{repTail}\ (\mathsf{Bin}\ t\ u)\ \mathit{xs} = \textbf{let}\ (t', \mathit{xs}') = \mathit{repTail}\ t\ \mathit{xs}\\
&\qquad\qquad\qquad\qquad\qquad\quad (u', \mathit{xs}'') = \mathit{repTail}\ u\ \mathit{xs}'\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathsf{Bin}\ t'\ u', \mathit{xs}'')\ .
\end{aligned}
$$

確實如同所預期的，串列 xs 被 $repTail\ t$ 使用，得到標籤過的新樹 t', 和剩下的串列 xs'. 後者再被 $repTail\ u$ 用來給 u 上標籤。最後我們得傳回剩下的串列 xs''. 實際上把串列變短的動作發生在基底狀況 $repTail\ (\mathsf{Tip}\ \_)$ 中。串列中的每個元素只會在每次遇見 Tip 時被取出一次，因此這是一個線性時間的演算法。

[todo: $repsort\ t = rep\ t\ (sort\ (leaves\ t\ [\,]))$.]

Burstall and Darlington [1977]

習題 5.13 — 回顧第 1.10 節中談到的 ITree:

$$\mathbf{data}\ \mathsf{ITree}\ a = \mathsf{Null} \mid \mathsf{Node}\ a\ (\mathsf{ITree}\ a)\ (\mathsf{ITree}\ a)\ .$$

猴麵包樹 *(baobab)*，又稱猢猻木，是一種樹幹相當粗的樹。[a] 如果一個 ITree Int 的每個標籤都大於其兩個子樹的標籤總和，我們便說它是一棵猴麵包樹。以下的函數判定一棵樹是否為猴麵包樹（其中 $sumT :: \mathsf{ITree\ Int} \to \mathsf{Int}$ 計算一個樹中所有標籤的總和）：

$$\begin{aligned}
&baobab :: \mathsf{ITree\ Int} \to \mathsf{Bool}\\
&baobab\ \mathsf{Null} &&= \mathsf{True}\\
&baobab\ (\mathsf{Node}\ x\ t\ u) &&= baobab\ t \wedge baobab\ u \wedge\\
&&&\quad x > (sumT\ t + sumT\ u)\ .
\end{aligned}$$

因反覆呼叫 $sumT$, 當樹的大小為 n 時，$baobab$ 的執行時間為 $O(n^2)$. 請使用組對的技巧，在 $O(n)$ 的時間內算出 $baobab$.

[a] 猴麵包樹原產於馬達加斯加、非洲等地，也被寫進了《小王子》之中。

習題 5.14 — 本題出自 Hu et al. [1997]。函數 $depth$ 定義一棵 ETree 的深度。Tip 的深度為零，Bin 的深度則為兩子樹中較深者的深度加一：

$$\begin{aligned}
&depth :: \mathsf{ETree}\ a \to \mathbb{N}\\
&depth\ (\mathsf{Tip}\ _) &&= \mathbf{0}\\
&depth\ (\mathsf{Bin}\ t\ u) &&= \mathbf{1}_+\ (depth\ t \uparrow depth\ u)\ .
\end{aligned}$$

下列函數 $deepest$ 則傳回一棵樹中最深的標籤：

$$\begin{aligned}
&deepest :: \mathsf{ETree}\ a \to \mathsf{List}\ a\\
&deepest\ (\mathsf{Tip}\ x) &&= [x]\\
&deepest\ (\mathsf{Bin}\ t\ u) &&\mid m < n = deepest\ u\\
&&&\mid m == n = deepest\ t +\!\!+ deepest\ u\\
&&&\mid m > n = deepest\ t\\
&\quad \mathbf{where}\ (m,n) &&= (depth\ t, depth\ u)\ .
\end{aligned}$$

請用組對的技巧，避免重複計算 $depth$. **注意**: 完成的程式中，$(+\!\!+)$ 仍可能需要 $O(n^2)$ 的時間。我們將在習題 5.27處理這個問題。

習題 5.15 — 第 3.2 節中的函數 $balanced :: \mathsf{RBTree} \to \mathsf{Bool}$ 檢查一棵紅黑樹是否平衡。由於重複呼叫 *bheight*, 這是一個需要 $O(n^2)$ 時間的函數。請用組對的技巧推導出一個可在線性時間內判斷平衡的版本。

5.5.3 代換為最小標籤 — 循環程式

下列函數 *minE* 曾出現在第 2.8 節中，找出一棵 ETree 中最小的值。函數 *rep* 則可視為 *repl* 的簡單版，將樹中的每個標籤都代換成同一個值 y。

$$\begin{aligned} &minE :: \mathsf{ETree\ Int} \to \mathsf{Int} \\ &minE\ (\mathsf{Tip}\ x) \quad = x \\ &minE\ (\mathsf{Bin}\ t\ u) = minE\ t \downarrow minE\ u \quad , \end{aligned} \qquad \begin{aligned} &rep :: b \to \mathsf{ETree}\ a \to \mathsf{ETree}\ b \\ &rep\ y\ (\mathsf{Tip}\ \_) = \mathsf{Tip}\ y \\ &rep\ y\ (\mathsf{Bin}\ t\ u) = \mathsf{Bin}\ (rep\ y\ t)\ (rep\ y\ u) \quad . \end{aligned}$$

給一棵樹 t, 我們要把 t 之中的每個標籤都代換成 t 的最小標籤。直觀的做法是寫成 $\mathbf{let}\ m = minE\ t\ \mathbf{in}\ rep\ t\ m$. 如此一來，$t$ 會被走訪兩次，第一次被 *minE* 走訪以計算 m, 第二次由 *rep* 進行代換。Bird [1984] 提出挑戰：有可能在只把 t 走訪一次的情況下，完成上述工作嗎？

以下函數 *repmin* 同時傳回代換過後的樹，以及原樹中的最小標籤：

$$\begin{aligned} &repmin :: \mathsf{ETree\ Int} \to a \to (\mathsf{ETree}\ a \times \mathsf{Int}) \\ &repmin\ y = \langle rep\ y, minE \rangle \quad . \end{aligned}$$

使用本節的技巧，讀者們應該已經可以為 *repmin* 導出如下的歸納定義：

$$\begin{aligned} &repmin\ y\ (\mathsf{Tip}\ x) \quad = (\mathsf{Tip}\ y, x) \\ &repmin\ y\ (\mathsf{Bin}\ t\ u) = \mathbf{let}\ (t', m) = repmin\ y\ t \\ &\qquad\qquad\qquad\qquad\quad\ (u', n) = repmin\ y\ u \\ &\qquad\qquad\qquad\quad \mathbf{in}\ (\mathsf{Bin}\ t'\ u', m \downarrow n) \quad . \end{aligned}$$

該定義只將 t 走訪一次。然後我們定義：

$$\begin{aligned} &transform :: \mathsf{ETree\ Int} \to \mathsf{ETree\ Int} \\ &transform\ t = \mathbf{let}\ (t', m) = repmin\ m\ t\ \mathbf{in}\ t' \quad . \end{aligned}$$

函數 *transform* 用 *repmin* 算出 t 的最小標籤 m, 同時又用 m 來標記 t... 於是，似乎確實用一次走訪就完成了兩件事！這是怎麼做到的呢？

函數 *transform* 有個特殊之處：變數 m 既是 *repmin* 的傳回值，又是其參數。這是一個循環程式 (*circular program*)。這樣的程式之所以能正常終止，有賴於 Haskell 的範式順序/惰性求值（見第 19頁）。實際上發生的事情如此：假設輸入為

$$\begin{aligned} t = \mathsf{Bin}\ &(\mathsf{Bin}\ (\mathsf{Tip}\ 4)\ (\mathsf{Tip}\ 2)) \\ &(\mathsf{Bin}\ (\mathsf{Bin}\ (\mathsf{Tip}\ 3)\ (\mathsf{Tip}\ 1))\ (\mathsf{Tip}\ 5)) \end{aligned}$$

repmin 將輸入 t 走訪一遍，邊走邊建立了一個未算出的算式 $(4 \downarrow 2) \downarrow ((3 \downarrow 1) \downarrow 5)$. 該算式就是 m 的值，其實也可視作一棵樹，其結構和 t 一樣，只是把每個 Bin

代換成 $(\downarrow)$. 函數 *repmin* 的另一項工作是新建一棵樹 t'，該樹之中每個 Tip 的標籤都指到這個算式。根據範式順序求值，這個算式還不用立刻被算出來。直到我們終於不得不算出它，例如當 t' 被傳回，我們要求電腦把 t' 印出來，或著有別的函數需檢查 t' 中的標籤時，該算式才被歸約成一個數字 — 這需要把該算式走訪一遍。(但，根據惰性求值，一但 m 被算成一個數字，下次使用 m 時就不再需要反覆計算了。)

需注意，雖然「t只被走訪一次」確實成立，這並不表示 *transform* 必然比老實將樹走訪兩次的程式快 — 效率是個複雜的問題，本章將陸續談到更多。我的建議是：循環程式應視為有趣、優雅的謎題，而不是以效率為目的的程式設計技巧。

5.5.4 小結與提醒

「組對」的技巧讓函數多傳回一些值。我們可能藉此省下一些重複的計算，增進效率。

我們自然想問：給一個有重複計算、待改進的函數 f, 怎知道該讓它多傳回什麼值？在本節的例子中，我們可由符號演算看出一些端倪：將 f 的定義展開，辨識出被重複計算的子算式，這些子算式就可能用來與原函數組對。但廣泛說來，「將一個函數或性質通用化」是編程與證明中最困難、最需要經驗、智慧的一步。只要找到正確的通用化，例如 *steepsum* 或 *repTail*, 剩下的推導都可相當機械化地進行。唯有「通用化」這一步，我們無法保證有任何機械化、公式化的方法可作為解決所有問題的萬靈丹 — 否則編程就是一件可完全自動化的事情了。

雖然如此，我們仍希望基於符號演算的形式方法能給我們一定程度的幫助與指引。程式語言研究的目標之一便是分辨出編程的過程中，哪些部分是瑣碎、機械化的，哪些部分是真正需要靈感與智慧的，並且盡量使用符號幫助，使我們在進行思考時有更多工具。

組對可用於減少走訪資料結構的次數，但這麼做並不見得有效率上的好處。例如，以下函數計算一個串列中元素的平均值：

$$average\ xs = sum\ xs \,/\, length\ xs \quad .$$

利用組對，我們可以另定義一個函數 $sumlen = \langle sum, length \rangle$, 並推導其歸納定義，在一次走訪中同時計算串列的和與長度：

$$\begin{aligned} sumlen\ [\,] &= (0,0) \\ sumlen\ (x:xs) &= \mathbf{let}\ (s,l) = sumlen\ xs \\ &\quad\ \mathbf{in}\ (x+s, 1+l) \quad . \end{aligned}$$

然後平均便可定義成 $average'\ xs = \mathbf{let}\ (s,l) = sumlen\ xs\ \mathbf{in}\ s \,/\, l$.

然而，根據 Hu et al. [1997]，$average'$ 通常比 *average* 慢。兩者都是 $O(n)$ 的演算法。函數 *sumlen* 在傳回值時會產生一個序對，該序對立刻被上層拆掉。因此 $average'$ 每處理一個元素耗費的時間較多，往往不如乾脆將 xs 走訪兩次。Hu et al. [1997] 認為需有更有效率的序對實作法，組對才是值得做的轉換。

習題 5.16 — 函數 *allpairs* 傳回輸入串列中任兩個元素（依其原本順序）形成的序對。例如 *allpairs* $[1,2,3,4]$ 可得到 $[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]$。

$$\begin{aligned}&allpairs :: \mathsf{List}\ a \to \mathsf{List}\ (a \times a)\\&allpairs\ [\,] \quad\ = [\,]\\&allpairs\ (x : xs) = map\ (\lambda y \to (x,y))\ xs +\!\!+ allpairs\ xs\ \ .\end{aligned}$$

而 *maxdiff* 則計算一個串列中任兩元素前者與後者的最大差：

$$\begin{aligned}&maxdiff :: \mathsf{List}\ \mathsf{Int} \to \mathsf{Int}\\&maxdiff = maximum \cdot map\ (\lambda(x,y) \to x - y) \cdot allpairs\ \ .\end{aligned}$$

當輸入串列長度為 n, 如此定義的 *maxdiff* 是一個需時 $O(n^2)$ 的演算法。定義：

$$mdm\ xs = (maxdiff\ xs, ???)\ \ .$$

找出 $???$ 可能的值，使得 *mdm* 能在 $O(n)$ 時間之內完成計算。你可假設以下性質：

$$maximum\ (map\ (x-)\ xs) = x - minimum\ xs\ \ . \tag{5.4}$$

並假設 *maximum* 與 *minimum* 在空串列上的值分別為 $-\infty$ 與 ∞.

5.6 累積參數

在第 5.2 節中，我們曾提及該處定義的 $reverse :: \mathsf{List}\ a \to \mathsf{List}\ a$ 需要 $O(n^2)$ 的時間。是否有比較快的作法呢？

5.6.1 串列反轉與連接

下列函數 *revcat* 似乎比 *reverse* 更通用一些：它拿兩個參數 *xs* 與 *ys*, 不僅將 *xs* 反轉，還把 *ys* 接到反轉後結果的右邊：[5]

$$\begin{aligned}&revcat :: \mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a\\&revcat\ xs\ ys = reverse\ xs +\!\!+ ys\ \ .\end{aligned}$$

原有的 *reverse* 可以視為 *revcat* 的特例 — $reverse\ xs = revcat\ xs\ [\,]$. 這裡是否也會出現責任越大，能力越強的現象：看似做了較多事情的 *revcat* 其實反倒有比較有效率的實作呢？我們試著演算看看！當 $xs := [\,]$, 顯然 $revcat\ [\,]\ ys = ys$. 考慮 $xs := x : xs$ 的情況：

$$\begin{aligned}&revcat\ (x : xs)\ ys\\=\ \ &\{\ revcat\ 之定義\ \}\end{aligned}$$

[5] *revcat* 為 'reverse" 與 'concat" 的簡寫。

$$
\begin{aligned}
& reverse\ (x:xs) +\!\!+ ys \\
= & \quad \{ reverse \text{ 之定義} \} \\
& (reverse\ xs +\!\!+ [x]) +\!\!+ ys \\
= & \quad \{ (+\!\!+) \text{ 之結合律} \} \\
& reverse\ xs +\!\!+ ([x] +\!\!+ ys) \\
= & \quad \{ (+\!\!+) \text{ 與 } revcat \text{ 之定義} \} \\
& revcat\ xs\ (x:ys)\ .
\end{aligned}
$$

因此我們有了：

$$
\begin{aligned}
& revcat\ [\,]\ \ \ \ \ ys = ys \\
& revcat\ (x:xs)\ ys = revcat\ xs\ (x:ys)\ .
\end{aligned}
$$

我們看看 $revcat\ [1,2,3,4]\ [\,]$ 如何歸約：

$$
\begin{aligned}
& \ \ revcat\ (1:2:3:4:[\,])\ [\,] \\
& = revcat\ (2:3:4:[\,])\ (1:[\,]) \\
& = revcat\ (3:4:[\,])\ (2:1:[\,]) \\
& = revcat\ (4:[\,])\ (3:2:1:[\,]) \\
& = revcat\ [\,]\ (4:3:2:1:[\,]) \\
& = [4,3,2,1]\ .
\end{aligned}
$$

串列 $[1,2,3,4]$ 中的元素一個個被搬到 $revcat$ 的第二個參數中。這是一個常數時間內可完成的動作，而每個元素只會被搬動一次。因此當 xs 的長度是 n，$reverse\ xs\ ys$ 可在 $O(n)$ 的時間內執行完畢！

我們剛看到的技巧和第 5.5 節中的組對互為對偶。在第 5.5 節中，我們為了導出一個函數的較快版本，讓它多傳回一些資訊。而在本節，我們則讓一個函數多吃些參數，多接受些資訊。通常這些新介紹的參數用於「累積」計算到目前為止的結果，例如在 $revcat$ 中，參數 ys 存放被反轉到一半的串列。因此這個技巧被稱作累積參數 (*accumulating parameters*)。

很多情況下，累積參數的運用仰賴某些運算元的結合律。觀察 $revcat$ 的推導，關鍵的一步便是 $(+\!\!+)$ 的結合律。它使我們能夠把 $(reverse\ xs +\!\!+ [x]) +\!\!+ ys$ 轉換為 $reverse\ xs +\!\!+ ([x] +\!\!+ ys)$，將 $[x]$ 往右搬，才能收回、累積到 $revcat$ 的第二個參數 ys 中。

回顧起來，我們最初如何發明出 $revcat\ xs\ ys = reverse\ xs +\!\!+ ys$ 這樣的定義呢？一個解釋便是：$reverse$ 之所以慢，是因為 $(+\!\!+)$ 的括號都括錯了方向，往左邊括了。因此我們在右邊補一個 $(+\!\!+ys)$, 希望用 $(+\!\!+)$ 的結合律，將括號往右挪。

我們多看一個例子。以下函數 $tags$ 傳回一棵 ITree 中所有的標籤：

$$
\begin{aligned}
& tags :: \mathsf{ITree}\ a \to \mathsf{List}\ a \\
& tags\ \mathsf{Null}\ \ \ \ \ \ \ \ \ \ = [\,] \\
& tags\ (\mathsf{Node}\ x\ t\ u) = tags\ t +\!\!+ [x] +\!\!+ tags\ u\ .
\end{aligned}
$$

和習題 5.7 的情況類似，當 t 是一棵向左傾斜的樹，$tags$ 會需要 $O(n^2)$ 的時間。例如，考慮這棵樹 t （以下將 $\mathsf{Node}\ x\ \mathsf{Null}\ \mathsf{Null}$ 簡寫為 $lv\ x$）：

$$t = \mathsf{Node}\ 7\ (\mathsf{Node}\ 6\ (\mathsf{Node}\ 4\ (\mathsf{Node}\ 2\ (lv\ 1)\ (lv\ 3))\ (lv\ 5)))\ (lv\ 8)\ .$$

將 $tags\ t$ 展開（並為說明方便，將一些 $[\,] +\!\!+ [x] +\!\!+ [\,]$ 化簡為 $[x]$），我們會得到：

$$((([1] +\!\!+ [2] +\!\!+ [3]) +\!\!+ [4] +\!\!+ [5]) +\!\!+ [6]) +\!\!+ [7] +\!\!+ [8]\ .$$

這個式子的結構和 t 一樣，只是將 $\mathsf{Node}\ x\ldots$ 都變為 $..+\!\!+[x]+\!\!+..$。我們可看到 $(+\!\!+[6])$ 需要走過一個長度為 5 的串列，$(+\!\!+[7])$ 需要走過一個長度為 6 的串列。

為了避免 $(+\!\!+)$ 的重複走訪，我們在 $tags\ t$ 的左邊補一個 $(+\!\!+ys)$，希望透過結合律改變括號的順序。我們定義：

$$\begin{aligned} &tagsAcc :: \mathsf{ITree}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a \\ &tagsAcc\ t\ ys = tags\ t +\!\!+ ys\ . \end{aligned}$$

利用 $(+\!\!+)$ 的結合律，讀者應不難導出如下的歸納定義：

$$\begin{aligned} &tagsAcc\ \mathsf{Null}\ \quad\quad ys = ys \\ &tagsAcc\ (\mathsf{Node}\ x\ t\ u)\ ys = tagsAcc\ t\ (x : tagsAcc\ u\ ys)\ . \end{aligned}$$

之後我們便可重新定義 $tags\ t = tagsAcc\ t\ [\,]$。至於 $tagsAcc$ 的效率如何呢？如果將 $tagsAcc\ t\ ys$ 展開，我們順利地得到：

$$1 : (2 : (3 : (4 : (5 : (6 : (7 : (8 : ys)))))))\ .$$

習題 5.17 — 由 $tagsAcc\ t\ ys = tags\ t +\!\!+ ys$ 推導出上述的歸納定義。

習題 5.18 — 確認 $tagsAcc\ t\ ys$ 確實是 $1 : (2 : (3 : (4 : (5 : (6 : (7 : (8 : ys)))))))$.

習題 5.19 — 接續習題 5.7，利用累積參數，推導出一個只需線性時間的 $tips :: \mathsf{ETree}\ a \to \mathsf{List}\ a$.

5.6.2 尾遞迴

關於第 5.6.1 節的 $revcat$ 有許多面向可談。本節先用它帶出一個重要的觀念：尾遞迴。

從第 2 章起，我們常見的歸納定義形式如下：

$$\begin{aligned} &f\ [\,] \quad\ = \ldots \\ &f\ (x : xs) = \ldots f\ xs \ldots\ . \end{aligned} \qquad\qquad \begin{aligned} &sum\ [\,] \quad\ = 0 \\ &sum\ (x : xs) = x + sum\ xs\ . \end{aligned}$$

右側的 sum 函數便是一個典型的例子。我們通常必須對遞迴呼叫的結果做些加工。例如此處 $sum\ xs$ 的結果需要經過 $(x+)$，才成為 $sum\ (x : xs)$ 的值。實作上，電腦在執行 $sum\ xs$ 的程式碼前必須記下「$sum\ xs$ 回傳後，必須回來執行 $(x+)$」這件事。這可能用堆疊或著其他方式達成。無論如何，當 xs 長度為 n，便有大

約 n 個這種「尚待完成的計算」被暫存著。當 xs 被走訪到了基底狀況，這些暫存的計算才一個個被收回。

但 *revcat* 的情況卻有點不同。觀察其程式碼，會發現遞迴呼叫 $revcat\ xs\ (x:ys)$ 的結果本身就是其左手邊 $revcat\ (x:xs)\ ys$ 的值，不需其他的加工：

$$\begin{aligned} &revcat\ []\ \ \ \ \ \ ys = ys \\ &revcat\ (x:xs)\ ys = revcat\ xs\ (x:ys)\ \ . \end{aligned}$$

因此，*revcat* 的實作中，遞迴呼叫完成後，電腦不需回到原呼叫之處再執行什麼東西：最後的結果 ys 可直接傳回到最上層、第一個呼叫 *revcat* 的地方！[6]

當一個函數 f 呼叫函數 g 時，如果該呼叫本身就是函數 f 最後的結果，並沒有針對傳回值的額外計算，我們將它稱之為一個尾呼叫 (*tail call*)。此名稱的由來可能是因為該呼叫是一連串計算後「最尾端」的動作。如果這是一個遞迴呼叫，則稱之為尾遞迴 (*tail recursion*)。

尾遞迴與迴圈　函數語言中的尾遞迴程式和指令式語言中的迴圈有相當密切的關係。回顧 $revcat\ [1,2,3,4]\ []$ 的歸約過程：

$$\begin{aligned} &\ \ \ revcat\ (1:2:3:4:[])\ [] \\ &= revcat\ (2:3:4:[])\ (1:[]) \\ &= revcat\ (3:4:[])\ (2:1:[]) \\ &= revcat\ (4:[])\ (3:2:1:[]) \\ &= revcat\ []\ (4:3:2:1:[]) \\ &= [4,3,2,1]\ , \end{aligned}$$

其實看來就像是一個有兩個變數的迴圈，其中一個由 $1:2:3:4:[]$ 逐漸縮短為 $[]$，另一個由 $[]$ 逐步增長為 $4:3:2:1:[]$. 函數 *reverse* 的定義 $reverse\ xs = revcat\ xs\ []$ 就是為這兩個變數設定初始值：如果要計算 xs 的反轉，兩變數應該分別初始化為 xs 與 $[]$. 在 *revcat* 的定義中，$revcat\ []\ ys$ 表示該迴圈在第一個變數為 $[]$ 時終止，此時程式傳回 ys；而 $revcat\ (x:xs)\ ys = revcat\ xs\ (x:ys)$ 則指定了在迴圈的每一步中兩個變數的值如何改變。函數 *reverse* 與 *revcat* 的組合相當於是這樣的一個指令式語言迴圈（假設 xs 與 ys 為變數，XS 為欲反轉的串列）：[7]

$$\begin{aligned} &xs, ys := \mathsf{XS}, [\,]; \\ &\mathbf{do}\ xs \neq [\,] \rightarrow \\ &\quad xs, ys := tail\ xs, head\ xs : ys \\ &\mathbf{od}; \\ &return\ ys \end{aligned}$$

而該迴圈的恆式 (loop invariant) 是什麼呢？正是 $reverse\ \mathsf{XS} = reverse\ xs +\!\!+ ys$ — 右手邊就是 *revcat* 的定義。

[6] 如果該語言的實作確實有做到這點，我們說它實作了尾呼叫消除 (*tail call elimination*) 或尾呼叫最佳化 (*tail call optimisation*)。有些語言到了蠻晚的版本才支援這個最佳化。

[7] $xs, ys := e_1, e_2$ 將 e_1 與 e_2 兩個值同時給予 xs 與 ys; **do** B → S **od** 表示一個迴圈，當 B 還成立時便反覆執行 S.

一般說來，一個尾遞迴函數可視為一個迴圈，其參數就是迴圈中的變數。遞迴呼叫時參數改變，相當於更新這些變數的值，其基底狀況則相當於迴圈的終止條件。當我們為了計算某個函數 f 而設計了另一個較通用、可寫成尾遞迴的函數 $fAcc$, $fAcc$ 的定義 (例如 $reverse\ xs +\!\!+ ys$) 往往就是這個迴圈的恆式 (*loop invariant*)。在指令式程式推導的方法學中，設計一個迴圈最難之處便是決定其恆式。如第 5.5.4 節所言，一般來說，並沒有一個固定的方法讓我們由 f 找出 $fAcc$。但我們仍可讓符號推演幫我們一些忙，並歸納出一些常見的模式。

串列總和 累積參數的技巧常被用來推導尾遞迴程式（雖然累積參數的應用並不僅止於此）。我們多看一個例子。如前所述，sum 函數每次被遞迴呼叫時會需要用堆疊或其他等價的方式記住每個 $(x+)$。是否能避免這麼做呢？我們定義：

$$\begin{aligned}&sumAcc :: \mathsf{List\ Int} \to \mathsf{Int} \to \mathsf{Int}\\ &sumAcc\ xs\ y = y + sum\ xs\quad .\end{aligned}$$

如此定義 $sumAcc$ 的直覺理由是：我們想用參數 y 存放累積的結果，希望利用 $(+)$ 的結合律逐步把 $sum\ xs$ 的結果搬入 y 中。同樣地，一但找到了這個定義，之後的演算便相當日常、例行了。我們先記下：$sum\ xs = sumAcc\ xs\ 0$. 接下來試圖導出 $sumAcc$ 的歸納定義。在基底情況中，我們會需要 $y+0 = y$。為了目睹結合律如何運作，我們看看 $xs := x : xs$ 的狀況：

$$\begin{aligned}&\quad sumAcc\ (x:xs)\ y\\ &= y + sum\ (x:xs)\\ &= y + (x + sum\ xs)\\ &= \quad \{\,(+)\ \text{之結合律}\,\}\\ &\quad (y+x) + sum\ xs\\ &= sumAcc\ xs\ (y+x)\quad .\end{aligned}$$

讀者可能發現我們之所以定義 $sumAcc\ xs\ y$ 為 $y + sum\ xs$, 而不是 $sum\ xs + y$, 是為了在使用結合律的那一步之中讓 x 可以就近與 y 結合。當然，$(+)$ 也滿足交換律，因此兩個定義其實是一樣的。當我們處理沒有交換律的運算元時，就得對這類細節更小心了。總之，我們可導出以下的程式：

$$\begin{aligned}&sumAcc\ [\,]\ \ \ \ \ \ y = y\\ &sumAcc\ (x:xs)\ y = sumAcc\ xs\ (y+x)\quad .\end{aligned}$$

這可翻譯成下述的兩種迴圈之一。左手邊的程式將串列 XS 加總，使用的兩個變數 xs 與 y 對應到 $sumAcc$ 的兩個參數。在右邊的程式中，我們假設輸入資料 $[x_0, x_1 \,..\, x_{N-1}]$ 依序被存放在一個陣列 X 中。陣列有 N 個元素，其索引範圍為 $[0\,..\,\mathsf{N}-1]$. 我們把陣列視為索引到內容的函數，因此 $\mathsf{X}\ i$ 存放著 x_i.

$$\begin{array}{l}xs, y := \mathsf{XS}, 0;\\ \textbf{do}\ xs \neq [\,] \to y\ \ := y + head\ xs;\\ \qquad\qquad\qquad\ \ xs := tail\ xs\\ \textbf{od};\\ return\ y\end{array}$$

$$\begin{array}{l}i, y := 0, 0;\\ \textbf{do}\ i \neq \mathsf{N} \to y := y + \mathsf{X}\ i;\\ \qquad\qquad\quad\ \ i := i + 1\\ \textbf{od};\\ return\ y\end{array}$$

確實，這就是我們一般用指令式語言加總一個串列/陣列時的常見迴圈寫法：由左到右走訪，用一個變數存放目前為止的和。以往許多人可能都沒注意到：同樣是「將 xs 由左到右走一遍」，這個程式和 $sum\ xs$ 是不同的演算法！$sum\ [1,2,3,4]$ 算出的是 $1+(2+(3+(4+0)))$, 而上述的、我們常用的那個迴圈算出的是 $(((0+1)+2)+3)+4$. 多虧 $(+)$ 的結合律，它們剛好是一樣的。

在迴圈中處理串列 在本節的結尾，我們更完整地討論一下歸納式的串列處理與迴圈的關係。初次閱讀的讀者可跳過本段。假設某函數 $f :: \mathsf{List\ A} \to \mathsf{B}$ 能寫成如下的形式：

$$\begin{aligned} f\ [\,] &= e \\ f\ (x : xs) &= x \oplus f\ xs \quad , \end{aligned}$$

其中 $e :: \mathsf{B}$, $(\oplus) :: \mathsf{A} \to \mathsf{B} \to \mathsf{B}$, 此處不假設 $(\oplus)$ 滿足結合律。我們能用一個尾遞迴函數（或著說用一個迴圈）計算 f 嗎？直覺上，我們可用一個迴圈將串列從右往左走一遍，並用變數紀錄目前為止算出的值。確實，如果我們要求如下的規格：

$$loop\ xs\ (f\ ys) = f\ (xs +\!\!+ ys) \quad , \tag{5.5}$$

（所以 $f\ xs = loop\ xs\ (f\ [\,]) = loop\ xs\ e$）並分析 $xs := [\,]$ 和 $xs := xs +\!\!+ [x]$ 的情況如下：

$$\begin{aligned} &loop\ [\,]\ (f\ ys) \\ =\ &\{\ loop\ 之規格\ \} \\ &f\ ([\,] +\!\!+ ys) \\ =\ &\{\ (+\!\!+)\ 之定義\ \} \\ &f\ ys \quad , \end{aligned}$$

$$\begin{aligned} &loop\ (xs +\!\!+ [x])\ (f\ ys) \\ =\ &\{\ loop\ 之規格\ \} \\ &f\ (xs +\!\!+ [x] +\!\!+ ys) \\ =\ &\{\ (+\!\!+)\ 之結合律，loop\ 之規格\ \} \\ &loop\ xs\ (f\ (x : ys)) \\ =\ &loop\ xs\ (x \oplus f\ ys) \quad . \end{aligned}$$

可知如下定義的 $loop$ 能滿足 (5.5)：

$$\begin{aligned} loop\ [\,]\ z &= z \\ loop\ (xs +\!\!+ [x])\ z &= loop\ xs\ (x \oplus z) \quad . \end{aligned}$$

但從串列尾端取出元素較不方便。如果我們改用下式作為規格：

$$loop\ xs\ (f\ ys) = f\ (reverse\ xs +\!\!+ ys) \quad , \tag{5.6}$$

可推導出

$$\begin{aligned} loop\ [\,]\ z &= z \\ loop\ (x : xs)\ z &= loop\ xs\ (x \oplus z) \quad . \end{aligned}$$

但此時要用 $loop$ 計算出 $f\ xs$，得先將 xs 反轉：$f\ xs = loop\ (reverse\ xs)\ e$. 這具體印證了我們的觀察：在串列上歸納定義出的函數，處理元素的順序和通常寫法的迴圈是相反的。

假設有 N 個元素的串列 $xs = [x_0, x_1 \ldots x_{N-1}]$ 被反過來存放在陣列 X 之中，意即 $\mathsf{X}\,0 = x_{N-1}, \mathsf{X}\,1 = x_{N-2}, \ldots \mathsf{X}\,(n-1) = x_0$. 前述的兩種 *loop* 函數都可在不同的詮釋下理解為如下的指令式程式。

$$
\begin{array}{l}
i, z := 0, e; \\
\textbf{do}\ i \neq \mathsf{N} \to z := \mathsf{X}\,i \oplus z; \\
\qquad\qquad i := i + 1 \\
\textbf{od}; \\
\mathit{return}\ z\ .
\end{array}
$$

5.6.3 更多尾遞迴範例

我們多看些使用結合律推導出尾遞迴程式，並增進效率的例子。

快速乘冪 說到結合律的應用，似乎不得不提作為經典例子的乘冪。我們在第 5.3.2 節中曾用二進位表示法導出一個用 $O(\log n)$ 個乘法計算 b^n 的程式。此處我們使用結合律推導出尾遞迴的版本。回顧函數 *exp* 在第 2.2 節中的定義，乘冪便是連續的乘法。我們定義以下的函數 *expAcc*，把 $\mathit{exp}\ b\ n$ 乘上一個累積參數 x，希望將一些中間結果搬移到 x 中：

$$
\begin{array}{l}
\mathit{expAcc} :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathit{expAcc}\ b\ n\ x = x \times \mathit{exp}\ b\ n\ .
\end{array}
$$

如果 *expAcc* 有快速的定義，我們可以令 $\mathit{exp}\ b\ n = \mathit{expAcc}\ b\ n\ 1$. 然後我們針對 n 為零、n 為非零的偶數，以及 n 為奇數三種情況作分析。當 $n := 0$, $\mathit{expAcc}\ b\ 0\ x = x$. 當 n 為偶數時，可以被改寫成 $2 \times n$. 以下的推導中我們將 $\mathit{exp}\ b\ n$ 寫成 b^n, 並假設它已有乘冪該有的各種性質：

$$
\begin{array}{ll}
 & \mathit{expAcc}\ b\ (2 \times n)\ x \\
= & x \times b^{2 \times n} \\
= & \quad \{ \text{因 } b^{m \times n} = (b^m)^n \} \\
 & x \times (b^2)^n \\
= & \quad \{ \mathit{expAcc} \text{ 之定義，} b^2 = b \times b \} \\
 & \mathit{expAcc}\ (b \times b)\ n\ x\ .
\end{array}
$$

當 n 是奇數，我們將它改寫成 $1 + n$:

$$
\begin{array}{ll}
 & \mathit{expAcc}\ b\ (1 + n)\ x \\
= & x \times b^{1+n} \\
= & \quad \{ \mathit{exp} \text{ 之定義} \} \\
 & x \times (b \times b^n) \\
= & \quad \{ (\times) \text{ 之結合律} \} \\
 & (x \times b) \times b^n \\
= & \quad \{ \mathit{expAcc} \text{ 之定義} \} \\
 & \mathit{expAcc}\ b\ n\ (x \times b)\ .
\end{array}
$$

將語法改寫成 Haskell 能接受的形式（例如將 $2 \times n$ 與 n 改寫成 n 與 $n \text{`}div\text{`} 2$）之後，我們得到這樣的程式：

$$\begin{aligned} &expAcc\ b\ 0\ x = x \\ &expAcc\ b\ n\ x \mid even\ n = expAcc\ (b \times b)\ (n \text{`}div\text{`} 2)\ x \\ &\qquad\qquad \mid odd\ \ n = expAcc\ b\ (n-1)\ (x \times b)\ \ . \end{aligned}$$

確實，這是一般在指令式語言中快速計算乘冪的方式。一個操作性的理解法是：$expAcc\ b\ n\ x$ 開始執行後，第一個參數中總是存放著 b 的「2 的某個次方」的乘冪（b, b^2, b^4...）。只在 n 是奇數時，當時的 b 才會被乘入累積參數 x 之中。

習題 5.20 — 本節的 $expAcc$ 函數相當於怎樣的指令式語言迴圈？其迴圈恆式為何？

習題 5.21 — 請推導出一個尾遞迴版本的 $length$ 函數。

習題 5.22 — 第 2.3 節中介紹了經典的階層函數：

$$\begin{aligned} &fact :: \mathbb{N} \to \mathbb{N} \\ &fact\ \mathbf{0} \qquad = 1 \\ &fact\ (\mathbf{1}_+\ n) = (\mathbf{1}_+\ n) \times fact\ n\ \ . \end{aligned}$$

請推導出一個尾遞迴版本。您利用了關於 $(\times)$ 的什麼性質？最後的程式是否像是一個計算 n 階層的指令式語言迴圈呢？

習題 5.23 — 第 2.2 節中把乘法定義為連續的加法。確實，一些早期、簡單的微電腦中沒有專做乘法的電路，只能以加法實作乘法。但如果有乘以二、除以二、以及判斷一個數字是奇數或偶數的指令（都是簡單的位元運算），我們只需 $O(\log m)$ 個加法即可計算 $m \times n$. 定義：

$$mulAcc\ m\ n\ k = k + m \times n\ \ .$$

請推導出一個只使用加減法、乘以二、除以二、奇偶判斷，並在 $O(\log m)$ 的時間內算出 $mulAcc\ m\ n\ k$ 的歸納定義。這個定義能被改寫成一個迴圈嗎？其恆式為何？

習題 5.24 — 下述函數 $dtoN$ 將一個以串列表達的十進位數字轉成自然數。例如 $dtoN\ [4,1,6,0] = 4160$:

$$\begin{aligned} &dtoN :: \mathsf{List}\ \mathbb{N} \to \mathbb{N} \\ &dtoN\ [\,] \qquad = 0 \\ &dtoN\ (d : ds) = d \times 10^{length\ ds} + dtoN\ ds\ . \end{aligned}$$

其中 $length\ ds$ 的反覆計算使得 $dtoN$ 成為一個需時 $O(n^2)$ 的演算法（n 為輸入串列的長度）。

1. 使用組對的技巧，導出一個能在 $O(n)$ 時間內計算 $dtoN$ 以及一些其他輔助結果的函數。
2. 使用累積參數，導出一個能在 $O(n)$ 時間內計算 $dtoN$ 的尾遞迴函數。**提示**：可試試看用這樣的定義 $dtoNAcc\ ds\ n = ..n.. + dtoN\ ds$.

多個累積參數 我們以一個稍微複雜的例子結束本節。給定以下函數：[8]

$$\begin{aligned} &masc :: \mathbb{N} \to \mathbb{N} \\ &masc\ 0 \qquad\quad = 1 \\ &masc\ (2 \times n) \quad = 2 \times masc\ n \\ &masc\ (1 + 2 \times n) = n + masc\ n\ . \end{aligned}$$

我們想用累積參數的技巧，推導出一個以尾遞迴計算 $masc\ n$ 的演算法。最難的一步總是尋找一個合適的通用化。我們試著展開 $masc$, 看看是否能找到什麼規律。以 $masc\ 43$ 為例：

$$\begin{aligned} & masc\ 43 \\ = {} & 21 + masc\ 21 \\ = {} & 21 + 10 + masc\ 10 \\ = {} & 31 + masc\ 10 \\ = {} & 31 + 2 \times masc\ 5 \\ = {} & 31 + 2 \times (2 + masc\ 2) \\ = {} & 35 + 2 \times masc\ 2 \\ = {} & 35 + 4 \times masc\ 1 \\ = {} & 35 + 4 \times 1 \\ = {} & 39\ . \end{aligned}$$

我們發現式子總能展開成為 $a + b \times masc\ n$ 的形式。因此我們定義

$$\begin{aligned} &mascAcc :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &mascAcc\ n\ a\ b = a + b \times masc\ n\ . \end{aligned}$$

但在開始推導 $mascAcc$ 前，我們得先確定 $masc$ 能由 $mascAcc$ 算得出來。幸好，我們可以讓 $masc\ n = mascAcc\ n\ 0\ 1$.

[8] $masc$ 為 “mostly ascending” 的縮寫。這個函數大致上遞增，但偶爾會掉下來一點點。

現在我們試著尋找 $mascAcc$ 的尾遞迴定義。當 $n := 0$, $mascAcc\ 0\ a\ b = a + b$. 當 n 為非零的偶數，可以將它代換為 $2 \times n$。我們推導：

$$\begin{aligned}
& mascAcc\ (2 \times n)\ a\ b \\
= {} & a + b \times masc\ (2 \times n) \\
= {} & \quad \{ masc \text{ 之定義} \} \\
& a + b \times 2 \times masc\ n \\
= {} & mascAcc\ n\ a\ (2 \times b)\ .
\end{aligned}$$

而當 n 為奇數，可以寫成 $1 + 2 \times n$ 的形式：

$$\begin{aligned}
& mascAcc\ (1 + 2 \times n)\ a\ b \\
= {} & a + b \times masc\ (1 + 2 \times n) \\
= {} & \quad \{ masc \text{ 之定義} \} \\
& a + b \times (n + masc\ n) \\
= {} & \quad \{ (\times) \text{ 分配入 } (+), (+) \text{ 之結合律} \} \\
& (a + b \times n) + b \times masc\ n \\
= {} & mascAcc\ n\ (a + b \times n)\ b\ .
\end{aligned}$$

因此我們有了如下的尾遞迴定義：

$$\begin{aligned}
& mascAcc\ 0\ a\ b = a + b \\
& mascAcc\ n\ a\ b \mid even\ n = mascAcc\ (n\ `div`\ 2)\ a\ (2 \times b) \\
& \qquad\qquad\quad \mid odd\ \ n = mascAcc\ (n\ `div`\ 2)\ (a + b \times (n\ `div`\ 2))\ b\ .
\end{aligned}$$

在推導 $mascAcc$ 的過程中使用了各種四則運算的性質，但總之目標是將式子整理回 $a + b \times masc\ n$ 的形式，以便收回成為 $mascAcc$。

習題 5.25 — 本題來自 Kaldewaij [1990]. 給定以下函數：

$$\begin{aligned}
& fusc :: \mathbb{N} \to \mathbb{N} \\
& fusc\ 0 \qquad\quad = 0 \\
& fusc\ 1 \qquad\quad = 1 \\
& fusc\ (2 \times n) \quad = fusc\ n \\
& fusc\ (1 + 2 \times n) = fusc\ n + fusc\ (1 + n)
\end{aligned}$$

請推導出一個計算 $fusc$ 的尾遞迴程式。**提示**: 以 $fusc\ 78$ 為例展開，看是否能找到什麼規律。

習題 5.26 — 我們能以尾遞迴的方式做 $map\ f$ 嗎？如果以 $mapAcc\ f\ xs\ ys = ys +\!\!+ map\ f\ xs$ 為規格，我們會導出如下的歸納定義：

$$\begin{aligned} mapAcc\ f\ []\ \ \ \ \ \ ys &= ys \\ mapAcc\ f\ (x:xs)\ ys &= mapAcc\ f\ xs\ (ys +\!\!+ [f\ x])\ . \end{aligned}$$

但重複的 $(+\!\!+[f\ x])$ 需要 $O(n^2)$ 的時間。請問用什麼樣的規格才能導出下面的歸納定義呢？

$$\begin{aligned} mapAcc\ f\ []\ \ \ \ \ \ ys &= reverse\ ys \\ mapAcc\ f\ (x:xs)\ ys &= mapAcc\ f\ xs\ (f\ x:ys)\ . \end{aligned}$$

5.6.4 尾遞迴的效率考量

如前一節所述，歸納定義的 *sum* 需用堆疊（或其他功能相當的機制）記下每個遞迴呼叫的結果該怎麼加工。這會佔用與輸入串列長度成正比的額外空間，相當不理想 — 直覺上，「將一個串列加總」應該是只需定量的額外空間即可完成的計算。尾遞迴的 *sumAcc* 則只需用一個變數存放目前為止的總和，當串列走訪到底，可直接將這個總和傳到最上層，似乎合理多了。因此，在以效率為考量的函數語言程式庫中，諸如加總、算最小值等等的函數幾乎都是以尾遞迴方式寫成的。大部分函數語言使用及早求值 — 在呼叫一個函數之前，總是把其參數先算成範式（見第 1.1節，第 19頁）。對這些語言來說，在上述場合使用尾遞迴函數確實只使用了定額的額外空間，效果相當好。

但對使用惰性求值的 Haskell 來說，情況又更複雜一些 — 若以分析工具實測，我們會發現純以本章的方式寫出的 *sumAcc* 仍用了和串列長度成正比的記憶體空間！這是怎麼回事呢？如前所述，$sumAcc\ [1,2,3,4...]\ 0$ 的值是 $((((0+1)+2)+3)+4)+...$。串列越長，這個式子越長。根據惰性求值的原則，Haskell 不拖到被強迫求值的最後一刻是不會將算式歸約的。因此在走訪串列的過程中，$0+1$ 不會歸約成 1，$(0+1)+2$ 不會歸約成 3... 這個大算式就這麼存放在記憶體中。直到整個串列被走訪完，*sumAcc* 的呼叫者要檢查其傳回值了（可能是要將它印出來，或著做樣式配對），這個大算式才又一步步被化簡成一個單一數值。

為改善這類情況下的效率，Haskell 提供了一些方法讓我們早點把一些數值強迫算成範式。例如在資料型別上標注某些欄位為「嚴格 (strict)」的、使用內建函數 *seq* 將數值歸約、或甚至使用更低階、屬於特定編譯器的「無盒型別 (unboxed type)」等等。對實務導向的 Haskell 編程員來說這些都是實用的技巧，只是已超出本書的範疇。

如果函數傳回的不是數值，而是結構化的資料，而程式語言支援惰性求值，情形又有所不同。如習題 5.26 中所見，我們可用尾遞迴的方式做 $map\ f$:

$$\begin{aligned} mapAcc\ f\ []\ \ \ \ \ \ ys &= reverse\ ys \\ mapAcc\ f\ (x:xs)\ ys &= mapAcc\ f\ xs\ (f\ x:ys)\ . \end{aligned}$$

乍看之下，這個程式的問題似乎是需要多做一次 *reverse*。但這可能並不很嚴重：*reverse* 也可用 *revcat* 實作，在線性時間內完成。該定義和原歸納定義的 *map f* 的最大差別是：*mapAcc f* 需等到輸入串列整個被走訪完畢後才會開始傳回第一個結果。而回顧 *map f* 的歸納定義：

$$\begin{aligned} &map\ f\ [\,] &&= [\,] \\ &map\ f\ (x : xs) &&= f\ x : map\ f\ xs \quad , \end{aligned}$$

歸納狀況中，$f\ x : \ldots$ 可在走訪到 x 時便先產生。考慮這樣的程式：$length \cdot filter\ p \cdot map\ f$。根據惰性求值，$f\ x : \ldots$ 會立刻被 *filter* 與 *length* 接收，然後才開始計算 $map\ f\ xs$. 因此 *map f* 回傳的中間串列其實並不會被完整地產生。在大部分情況下，歸納定義的 *map f* 是比尾遞迴的 *mapAcc f* 更有效率的函數。在許多情形中，早點產生部分的結果、早點讓它被使用掉，會是在空間與時間上都更有效率的做法。

5.6.5 函數作為串列

既然說到串列反轉，本節延伸介紹一個相關且相當有用的技巧。回顧 *revcat* 的尾遞迴定義，若將最後一個參數省去，其定義可改寫成：

$$\begin{aligned} &revcat :: \mathsf{List}\ a \to (\mathsf{List}\ a \to \mathsf{List}\ a) \\ &revcat\ [\,] &&= id \\ &revcat\ (x : xs) &&= revcat\ xs \cdot (x :) \quad . \end{aligned}$$

revcat 是一個傳回 $\mathsf{List}\ a \to \mathsf{List}\ a$ 的函數。在基底狀況，$revcat\ [\,]$ 傳回 *id*. 歸納狀況中，$revcat\ xs$ 傳回的函數和 $(x :)$ 組合在一起。這和 *reverse* 其實很像：在基底狀況，$reverse\ [\,]$ 傳回空串列 $[\,]$, 歸納狀況中，$reverse\ (x : xs)$ 傳回 $reverse\ xs +\!\!+ (x : [\,])$。好像把空串列代換成 *id*, 把 $(+\!\!+)$ 變成 $(\cdot)$, 我們就得到 *revcat* 了。

這令我們聯想：有沒有可能把 $\mathsf{List}\ a \to \mathsf{List}\ a$ 視為串列的另一種表示法，其中 *id* 就是空串列，而串列連接就是 $(\cdot)$ 呢？定義：

$$\mathbf{type}\ \mathsf{DList}\ a = \mathsf{List}\ a \to \mathsf{List}\ a \quad ,$$

一個型別為 $\mathsf{DList}\ a$ 的函數 f 表示一個「尾段尚未確定」的串列 — 餵給它一個尾巴 ys, $f\ ys$ 便會傳回一個真正的串列。若 xs 是一個型別為 $\mathsf{List}\ a$ 的串列，$(xs +\!\!+)$ 便是如此的一個 $\mathsf{DList}\ a$. 要將一個 $\mathsf{DList}\ a$ 轉成 $\mathsf{List}\ a$, 則只需將 $[\,]$ 傳進去即可。下列兩個函數幫我們在這兩種表示法之間作轉換：

$$\begin{aligned} &toDList :: \mathsf{List}\ a \to \mathsf{DList}\ a \\ &toDList\ xs = (xs +\!\!+) \quad , \end{aligned} \qquad \begin{aligned} &toList :: \mathsf{DList}\ a \to \mathsf{List}\ a \\ &toList\ xs = xs\ [\,] \quad . \end{aligned}$$

空串列被轉換成 $([\,] +\!\!+)$, 化簡一下之後確實得到了 *id*. 「含一個單一元素 x」的 DList 可寫成 $(x :)$ — 給任何串列，$(x :)$ 把 x 接到其最左邊。

兩個 $\mathsf{DList}\ a$ 如何連接在一起呢？使用函數組合 $(\cdot)$。確實，$(xs +\!\!+) \cdot (ys +\!\!+)$ 是一個接收任一個尾段 ws, 傳回 $xs +\!\!+ (ys +\!\!+ ws)$ 的函數。DList 版本的 'cons" 建構元則可定義成

$$\begin{aligned} &nil :: \mathsf{DList}\ a \\ &nil = id \ , \end{aligned} \qquad \begin{aligned} &cons :: a \to \mathsf{DList}\ a \to \mathsf{DList}\ a \\ &cons\ x\ xs = (x:) \cdot xs \ . \end{aligned}$$

List 上的 $(+\!\!+)$ 若往左結合，效率會較不好。例如 $(xs +\!\!+ ys) +\!\!+ zs$ 會需要把 xs 走訪兩次。DList 的情況呢？考慮 $((xs+\!\!+) \cdot (ys+\!\!+)) \cdot (zs+\!\!+)$，其中左邊的兩個 DList 被括在一起。我們演算看看給了一個尾端 ws 後的情況：

$$\begin{aligned} & ((xs+\!\!+) \cdot (ys+\!\!+)) \cdot (zs+\!\!+)\ \$\ ws \\ = \quad & \{\ (\cdot)\ 之定義\ \} \\ & (xs+\!\!+) \cdot (ys+\!\!+)\ \$\ zs +\!\!+ ws \\ = \quad & \{\ (\cdot)\ 之定義\ \} \\ & (xs+\!\!+)\ \$\ ys +\!\!+ (zs +\!\!+ ws) \\ = \ & xs +\!\!+ (ys +\!\!+ (zs +\!\!+ ws)) \ . \end{aligned}$$

由於 $(\cdot)$ 的定義，即使 $(xs+\!\!+) \cdot (ys+\!\!+)$ 先被組合在一起，我們仍得到括號往右括的 $xs +\!\!+ (ys +\!\!+ (zs +\!\!+ ws))$！

我們多研究一個例子。在習題 5.7 中，給定一個 ETree a, 我們想傳回其所有的標記。如果寫成 $tips\ (\mathsf{Bin}\ t\ u) = tips\ t +\!\!+ tips\ u$，當樹往左邊傾斜時，程式會需要 $O(n^2)$ 的時間。但如果我們改用 DList:

$$\begin{aligned} &tipsD :: \mathsf{ETree}\ a \to \mathsf{DList}\ a \\ &tipsD\ (\mathsf{Tip}\ x) \ \ = (x:) \\ &tipsD\ (\mathsf{Bin}\ t\ u) = tipsD\ t \cdot tipsD\ u \ . \end{aligned}$$

上述函數會邊走訪輸入的樹，邊產生一個結構與輸入樹相同的 DList。例如，考慮如下的樹：

$$\begin{aligned} t = \mathsf{Bin}\ &(\mathsf{Bin}\ (\mathsf{Bin}\ (\mathsf{Tip}\ 5)\ (\mathsf{Tip}\ 4)) \\ &\qquad (\mathsf{Bin}\ (\mathsf{Tip}\ 3)\ (\mathsf{Tip}\ 2))) \\ &(\mathsf{Tip}\ 1) \ , \end{aligned}$$

$tipsD\ t$ 將會是 $(((5:) \cdot (4:)) \cdot ((3:) \cdot (2:))) \cdot (1:)$ – 結構和 t 相同，只是將 t 的每個 Tip x 代換成 $(x:)$, 每個 Bin 代換成 $(\cdot)$. 但當我們傳一個空串列進去，這個由 $(x:)$ 和 $(\cdot)$ 形成的「樹」將被走訪一遍，並在線性時間內得到 $5:4:3:2:1:[\,]$。事實上，如果我們為 $tipsD$ 補一個參數並展開，我們將得到和習題 5.19 中一樣的結果。有了 $tipsD$, 我們可將原有的 $tips$ 改定義為 $tips = toList \cdot tipsD$, 或著 $tips\ t = tipsD\ t\ [\,]$.

Hughes [1986]

習題 5.27 — 在習題 5.14 中，我們使用組對可能得到一個型別為 ETree $a \to (\mathsf{List}\ a, \mathbb{N})$ 的函數。該函數仍需要 $O(n^2)$ 的時間連接串列（其中 n 為輸入樹的大小）。請設計一個 $O(n)$ 的演算法。

CHAPTER 6

摺

第 2章中的許多歸納函數定義都循著同一個固定模式。以 *sum*, *length*, 與 *map f* 為例：

$$
\begin{aligned}
&sum :: \mathsf{List}\ \mathsf{Int} \to \mathsf{Int} \\
&sum\ [\,] \quad\ = 0 \\
&sum\ (x : xs) = x + sum\ xs\ \ , \\
&length :: \mathsf{List}\ a \to \mathbb{N} \\
&length\ [\,] \quad\ = \mathbf{0} \\
&length\ (x : xs) = \mathbf{1}_{+}\ (length\ xs)\ \ , \\
&map :: (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b \\
&map\ f\ [\,] \quad\ = [\,] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs\ \ .
\end{aligned}
$$

它們都在輸入為 $[\,]$ 時傳回某個基底值，在輸入為 $x : xs$ 時在 xs 上遞迴呼叫，並將呼叫結果稍作加工。三者的不同之處只在橘色的部分，即基底值以及用於加工的函數：*sum* 使用 0 與 $(+)$, *length* 使用 $\mathbf{0}$ 與 $\mathbf{1}_{+}$, *map f* 則使用 $[\,]$ 與 $(f\ x :)$. 如果說「抽象化」是一個高階程式語言給我們的最重要能力，我們能否將這個模式抽象出來呢？

我們把上述三個定義中橘色的部分抽出變成參數，將餘下的函數稱為 *foldr*:

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to \mathsf{List}\ a \to b \\
&foldr\ f\ e\ [\,] \quad\ = e \\
&foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)\ \ .
\end{aligned}
$$

如此一來，*sum*, *length*, 與 *map f* 都是 *foldr* 的特例：

$$
\begin{aligned}
sum &= foldr\ (+)\ 0\ , \\
length &= foldr\ (\lambda x\ n \to \mathbf{1}_+\ n)\ \mathbf{0}\ , \\
map\ f &= foldr\ (\lambda x\ ys \to f\ x : ys)\ [\,]\ .
\end{aligned}
$$

函數 *foldr* 是串列上的「摺 (fold)」— *foldr* 一詞是 fold 與「右邊 (right)」的縮寫，意謂該函數是一個往右結合的摺。我們將在下一節解釋。

6.1 串列的摺

「摺」有許多方式可理解。我們可說 *foldr* 捕捉了最常見的一種歸納定義模式，並將它形式化地表達出來。在第 2章中，許多函數定義都遵循這樣的模式：

$$
\begin{aligned}
&h :: \mathsf{List}\ a \to b \\
&h\ [\,] \quad\ = e \\
&h\ (x : xs) = \ldots x \ldots h\ xs \ldots
\end{aligned}
$$

在 $h\ [\,]$ 的情況傳回某個基底值；在 $h\ (x : xs)$ 的情況中可使用 x 與 $h\ xs$ 的值。如果上述定義中 $\ldots$ 之處沒有出現 xs, 則 h 的定義就能寫成一個 *foldr*.

我們也可將 *foldr* 視為組件 (combinator) 函數之一。第 1.8.3節之中介紹了組件函數的觀念：如同 *map*, *take*, *drop*, *zip* 等等的組件函數捕捉了常見的程式設計模式。每個組件負責一項單一、通用、易重用的功能。函數 *foldr* 也可視為一個組件，只是它比一些其他組件更抽象、更通用 — 我們稍後將發現許多我們見過的組件函數都是 *foldr* 的特例。

還有一個理解 *foldr* 的方式：*foldr* 替換了串列中的建構元。回顧：任何有限長度的串列都是由 $[\,]$ 開始，有限次地套用 $(:)$ 而來。例如 $[x_0, x_1, x_2]$ 是 $x_0 : (x_1 : (x_2 : [\,]))$ 的簡寫。考慮 $foldr\ (\oplus)\ e\ [x_0, x_1, x_2]$:

$$
\begin{aligned}
&foldr\ (\oplus)\ e\ (x_0 : (x_1 : (x_2 : [\,]))) \\
&= x_0 \oplus foldr\ (\oplus)\ e\ (x_1 : (x_2 : [\,])) \\
&= x_0 \oplus (x_1 \oplus foldr\ (\oplus)\ e\ (x_2 : [\,])) \\
&= x_0 \oplus (x_1 \oplus (x_2 \oplus foldr\ (\oplus)\ e\ [\,])) \\
&= x_0 \oplus (x_1 \oplus (x_2 \oplus e))\ .
\end{aligned}
$$

我們可看到 *foldr* 將串列走訪一次，將每個 $(:)$ 替換成 $(\oplus)$, 將 $[\,]$ 替換成 e. 式子中的括號往右邊結合，這是 *foldr* 的名字中字母 r 的由來。這種理解也便於解釋 *foldr* 的型別。回想串列的兩個建構元，

- $[\,]$ 的型別是 $\mathsf{List}\ a$,
- $(:)$ 的型別是 $a \to \mathsf{List}\ a \to \mathsf{List}\ a$.

函數 $foldr\ (\oplus)\ e$ 接收一個 $\mathsf{List}\ a$，把其中的建構元分別替換為 e 與 $(\oplus)$，藉此算出一個型別為 b 的值。因此，

- e 是輸入為 $[\,]$ 時立刻傳回的值，其型別必須是 b.
- 至於 $(\oplus)$ 的型別，考慮 $x_0 \oplus (x_1 \oplus (x_2 \oplus e))$ 這個式子。其中 x_0 的型別為 a, $x_1 \oplus (x_2 \oplus e)$ 是建構元已被替換過的串列，型別應該為 b. 而 $(\oplus)$ 拿到這兩個輸入後，得算出一個型別為 b 的值. 因此 $(\oplus)$ 的型別為 $a \to b \to b$.

注意：e 與 $(\oplus)$ 的型別分別是將 $[\,]$ 與 $(:)$ 的型別中的 List a 代換成 b 而來。綜合言之，$foldr$ 的型別是 $(a \to b \to b) \to b \to \text{List } a \to b$.

為方便說明，此後我們將 $foldr\ (\oplus)\ e$ 之中的 e 稱作基底值 (*base value*)，將 $(\oplus)$ 稱作步驟函數 (*step function*). 函數 $foldr$ 的型別可以理解為：給一個型別為 $a \to b \to b$ 的步驟函數，和一個型別為 b 的基底值，$foldr$ 就能將一個 List a 轉換為 b.

串列的摺 (*foldr*) 只是一個常用的特例 —「將資料結構中的建構元代換掉」的動作也可推廣到其他資料結構上。我們在之後的章節中將看到一些其他資料結構上的摺。

下一節將舉更多使用 $foldr$ 的例子。在那之前我們再次提醒讀者：在 $foldr\ (\oplus)\ e\ (x : xs)$ 的狀況中，$(\oplus)$ 可以使用 x 與 $foldr\ (\oplus)\ e\ xs$ 的結果，但不能直接使用 xs.

6.1.1 更多串列上的摺

回顧起來，我們可發現第 2章介紹的許多函數都是 $foldr$.

例 6.1. 以下函數都可寫成 $foldr$:

- $concat = foldr\ (+\!\!+)\ [\,]$.
- $filter\ p = foldr\ (\lambda x\ xs \to \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)\ [\,]$,
- $takeWhile\ p = foldr\ (\lambda x\ xs \to \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ [\,])\ [\,]$,
- $elem\ x = foldr\ (\lambda y\ b \to x == y \vee b)$ False,
- $all\ p = foldr\ (\lambda x\ b \to p\ x \wedge b)$ True.

串列連接 $(+\!\!+) :: \text{List } a \to \text{List } a \to \text{List } a$ 雖是個二元運算，若將 $(+\!\!+ ys) :: \text{List } a \to \text{List } a$ 視為一個函數，它可寫成一個 $foldr$:

$$(+\!\!+ ys) = foldr\ (:)\ ys\ .$$

一個重要的特例是當 $ys = [\,]$ 時。對任何 xs, $xs +\!\!+ [\,] = xs$. 因此 $(+\!\!+ [\,])$ 是串列上的 id:

$$\begin{aligned} &id :: \text{List } a \to \text{List } a \\ &id = foldr\ (:)\ [\,]\ . \end{aligned}$$

確實，將一個串列中的 $(:)$ 代換成 $(:)$, $[\,]$ 代換成 $[\,]$, 我們還是得到原來的串列。我們日後還會用到「串列上的 id 是一個 $foldr$」的性質。

計算所有前段的函數 $inits :: \text{List } a \to \text{List } (\text{List } a)$ 可寫成 $foldr$:

$$inits = foldr\ (\lambda x\ xss \to [\,] : map\ (x:)\ xss)\ [[\,]]\ .$$

計算所有後段的 $tails :: \text{List } a \to \text{List } (\text{List } a)$ 也可以寫成 $foldr$，但需要用一個小性質。回顧其定義：

$$\begin{aligned} tails\ [\,] &= [[\,]] \\ tails\ (x : xs) &= (x : xs) : tails\ xs\ . \end{aligned}$$

乍看之下這不符合 *foldr* 的模式：參數 *xs* 出現在 $\ldots : tails\ xs$ 的左邊，但在 *foldr* 的模式中，*xs* 不能出現在遞迴呼叫之外。幸好 *tails* 有一個剛好在此有用的小特性：*tails xs* 傳回的所有後段中，第一個就是 *xs* 本身：$head\ (tails\ xs) = xs$. 因此我們可將 *tails* 寫成：

$$tails = foldr\ (\lambda x\ xss \to (x : head\ xss) : xss)\ [[]]\ .$$

由於 *tails* 永遠傳回非空串列，使用 *head xss* 是安全的。

習題 6.1 — 請將以下函數寫成 *foldr*:

1. $perms :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$ (見第 2.6.4節),
2. $sublists :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$ (見第 2.6.4節),
3. $splits :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a \times \mathsf{List}\ a)$ （見習題 2.27）。

6.1.2 不是 *foldr* 的函數

並非所有輸入為串列的函數都是 *foldr*. 最明顯的例子是 *tail*: 我們無法由 *x* 和 *tail xs* 算出 $tail\ (x : xs)$。例如，$tail\ [1,2,3] = [2,3]$，但 $tail\ [2,3] = [3]$, 而 $[2,3]$ 無法由 1 和 $[3]$ 組出來。

另一個例子是 *dropWhile p*. 回顧其定義：

$$\begin{aligned} dropWhile\ p\ [] &= [] \\ dropWhile\ p\ (x : xs) &= \mathbf{if}\ p\ x\ \mathbf{then}\ dropWhile\ p\ xs\ \mathbf{else}\ x : xs\ \ , \end{aligned}$$

在歸納情況中，**else** 的分支需傳回 $x : xs$ — *xs* 出現在遞迴呼叫以外的地方。這樣的程式不是 *foldr*。當然，這只表示 *dropWhile p* 的這個定義不符合 *foldr* 的模式. 是否有其他的方式能將 *dropWhile p* 寫成 *foldr* 呢？不論 *dropWhile p* 是怎麼定義的，考慮 $dropWhile\ even\ [4,3,6,2] = [3,6,2]$, 但 $dropWhile\ even\ [3,6,2] = []$ — 看來，*dropWhile p* 丟掉了太多資訊，使得我們無法保證能從 *dropWhile p xs* 重組出 $dropWhile\ p\ (x : xs)$。因此，*dropWhile p* 和 *tail* 一樣，是先天上無法寫成 *foldr* 的。

習題 6.2 — 考慮第 2.6.4節的函數 *fan*:

$$\begin{aligned} fan &:: a \to \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ fan\ y\ [] &= [[y]] \\ fan\ y\ (x : xs) &= (y : x : xs) : map\ (x :)\ (fan\ y\ xs)\ \ . \end{aligned}$$

為何這個定義目前的形式不是一個 *foldr*? 有沒有可能將 $fan\ y$ 寫成一個 *foldr* 呢？

6.2 摺融合定理

第 0章中提及，「抽象化」意指抽取出我們認為重要的概念、成分，給予一個名字或符號。如此一來，這個概念正式地「存在」了，我們可以談論它、研究其

性質，並將研究結果應用在所有符合這個抽象概念的事物上。一個程式語言最重要的功能之一是提供良好的抽象化機制。由於高階函數等等性質，函數語言讓我們能較容易地對程式結構作抽象。

「摺」是我們找到的一個抽象，許多程式可以表達為摺。而一旦辨識出了摺這個結構，我們可開始討論所有摺都滿足的性質，這些性質則將可適用於所有是摺的程式上。

關於摺的性質中，最重要的也許是本節的摺融合定理 (*fold-fusion theorem*)。

摺融合定理告訴我們一個摺如何能與串接於其後的函數融合起來，成為單獨的一個摺：

定理 6.2 (摺融合定理 (串列版)). 給定 $f :: a \to b \to b, e :: b, h :: b \to c$. 如果 $h\,(f\,x\,y) = g\,x\,(h\,y)$ 對所有 $x :: a$ 與在 $foldr\,f\,e$ 的值域中的 $y :: b$ 成立，則

$$h \cdot foldr\,f\,e = foldr\,g\,(h\,e)\ .$$

性質 $h\,(f\,x\,y) = g\,x\,(h\,y)$ 是該融合能成立的充分條件，我們日後將稱之為「融合條件 (*fusion condition*)」。如果定理本身看來太抽象，下述例子也許可給讀者一些直覺。考慮 $[x_0, x_1, x_2]$:

$$\begin{aligned}
& h\,(foldr\,f\,e\,[x_0, x_1, x_2]) \\
= \quad & \{\, foldr \text{ 之定義} \,\} \\
& h\,(f\,x_0\,(f\,x_1\,(f\,x_2\,e))) \\
= \quad & \{\, \text{融合條件: } h\,(f\,x\,y) = g\,x\,(h\,y) \,\} \\
& g\,x_0\,(h\,(f\,x_1\,(f\,x_2\,e))) \\
= \quad & \{\, \text{融合條件: } h\,(f\,x\,y) = g\,x\,(h\,y) \,\} \\
& g\,x_0\,(g\,x_1\,(h\,(f\,x_2\,e))) \\
= \quad & \{\, \text{融合條件: } h\,(f\,x\,y) = g\,x\,(h\,y) \,\} \\
& g\,x_0\,(g\,x_1\,(g\,x_2\,(h\,e))) \\
= \quad & \{\, foldr \text{ 之定義} \,\} \\
& foldr\,g\,(h\,e)\,[x_0, x_1, x_2]\ .
\end{aligned}$$

由此例可看出融合條件 $h\,(f\,x\,y) = g\,x\,(h\,y)$ 的作用 — 將 h 往右推，並將途中經過的 f 都變成 g, 直到碰到 e 為止。

定理 6.2可用例行的歸納證明證成：

Proof. 假設融合條件成立，我們需證明對所有 xs, $h\,(foldr\,f\,e\,xs) = foldr\,g\,(h\,e)\,xs$.

情況 $xs := [\,]$:

$$\begin{aligned}
& h\,(foldr\,f\,e\,[\,]) \\
= \; & h\,e \\
= \; & foldr\,g\,(h\,e)\,[\,]\ .
\end{aligned}$$

情況 $xs := x : xs$:

$$\begin{aligned}
& h\,(foldr\,f\,e\,(x : xs)) \\
= \quad & \{\, foldr \text{ 之定義} \,\}
\end{aligned}$$

$$
\begin{aligned}
& h\ (f\ x\ (foldr\ f\ e\ xs)) \\
= \quad & \{\text{融合條件: } h\ (f\ x\ y) = g\ x\ (h\ y)\} \\
& g\ x\ (h\ (foldr\ f\ e\ xs)) \\
= \quad & \{\text{歸納假設}\} \\
& g\ x\ (foldr\ g\ (h\ e)\ xs) \\
= \quad & \{foldr\ \text{之定義}\} \\
& foldr\ g\ (h\ e)\ (x : xs)\ .
\end{aligned}
$$

□

註記 我們在歸納情況的第二步使用了融合條件 $h\ (f\ x\ y) = g\ x\ (h\ y)$. 欲使該步成立，融合條件不須對所有 y 都成立 — 我們只需要它在 y 是 $foldr\ f\ e$ 的可能結果時成立即可。這是定理 6.2中「在 $foldr\ f\ e$ 的值域中的 y」這句話的由來。

在本章接下來大部分的例子中，我們其實可以證明融合條件對所有 y 均成立。但只要我們處理的演算法問題稍微複雜些，我們便會常遇到融合條件只對 $foldr\ f\ e$ 的值域中的 y 成立的情況。我們將在第 6.2.6節中看到一些例子。

6.2.1 將摺融合用於定理證明

定理 6.2有幾種用法：

- 一種可能是用於證明性質：我們希望證明 $h \cdot foldr\ f\ e$ 與 $foldr\ g\ (h\ e)$ 相等，此時我們已知 $h, f, g,$ 與 e.
- 另一種可能是用於生成程式。此時我們通常已知 $h, f,$ 與 e, 但不知道 g. 我們希望找到一個讓融合條件成立的 g, 使得 $h \cdot foldr\ f\ e$ 能在一個 $foldr$ 之中完成。

我們先討論第一種情況。

例 6.3. 回顧 map 融合定理 *(2.2)*: $map\ f \cdot map\ g = map\ (f \cdot g)$. 第 *63* 頁提供了一個歸納證明。由於 $map\ g$ 是一個摺，我們也可用摺融合定理證明如下。

$$
\begin{aligned}
& map\ f \cdot map\ g \\
= \quad & \{map\ \text{的摺定義}\} \\
& map\ f \cdot foldr\ (\lambda x\ ys \to g\ x : ys)\ [\,] \\
= \quad & \{\text{摺融合}\} \\
& foldr\ (\lambda x\ ys \to f\ (g\ x) : ys)\ [\,] \\
= \quad & \{map\ \text{的摺定義}\} \\
& map\ (f \cdot g)\ .
\end{aligned}
$$

第二步需要的融合條件只需簡單展開定義即可滿足：

$$
\begin{aligned}
& map\ f\ (g\ x : ys) \\
= \quad & \{map\ \text{之定義}\} \\
& f\ (g\ x) : map\ f\ ys\ .
\end{aligned}
$$

將上述例子與第 2.4節的歸納證明比較。歸納證明中，最關鍵的是「使用歸納步驟」的一步，而使用摺融合定理的證明卻沒有這步 — 歸納步驟的使用被包裝、隱藏在摺融合定理中了。而上述例子中關於融合條件的證明，恰巧是原歸納證明中和問題本身最相關的部分。

我們可說：摺融合定理之於證明，就如同摺之於程式。摺是抽象出的常見程式骨架，將拆解輸入串列、做遞迴呼叫等動作包裝起來。有了摺，我們不需自己做遞迴呼叫，只需填入針對特定問題的 f, e 等參數的值。摺融合定理則是抽象出的常見證明骨架，將狀況分析、使用歸納假設等動作包裝起來。有了摺融合定理，我們不需自己做狀況分析、引用歸納假設，只需填入針對這個問題的融合條件的證明。

事實上，*map* 融合定理是下述定理的特例：

定理 6.4 (*foldr*-*map* 融合定理). $foldr\ f\ e \cdot map\ g = foldr\ (f \cdot g)\ e$.

我們時常看到 *foldr* 與 *map* 一起出現，此時定理 6.4 相當好用。

例 6.5. 我們嘗試證明 $sum \cdot map\ (2\times) = (2\times) \cdot sum$. 首先考慮等號左手邊的 $sum \cdot map\ (2\times)$. 由於 *sum* 是一個 *foldr*, 我們可用定理 *6.4*將該式合併為一個 *foldr*:

$$
\begin{aligned}
& sum \cdot map\ (2\times) \\
= & \quad \{ sum \text{ 之摺定義} \} \\
& foldr\ (+)\ 0 \cdot map\ (2\times) \\
= & \quad \{ \text{定理 } 6.4\text{: } foldr\text{-}map \text{ 融合} \} \\
& foldr\ ((+) \cdot (2\times))\ 0\ .
\end{aligned}
$$

另一方面，$(2\times) \cdot sum$ 可以融合成同一個 *foldr*:

$$
\begin{aligned}
& (2\times) \cdot sum \\
= & \ (2\times) \cdot foldr\ (+)\ 0 \\
= & \quad \{ \text{摺融合} \} \\
& foldr\ ((+) \cdot (2\times))\ 0\ .
\end{aligned}
$$

其中的融合條件證明如下：

$$
\begin{aligned}
& 2 \times (x + y) \\
= & \quad \{ \text{乘法與加法之分配率} \} \\
& 2 \times x + 2 \times y \\
= & \quad \{ (\cdot) \text{ 之定義} \} \\
& ((+) \cdot (2\times))\ x\ (2 \times y)\ .
\end{aligned}
$$

由此我們證明了 $sum \cdot map\ (2\times) = (2\times) \cdot sum$.

許多等式可用類似的模式證明：為證明 $e_1 = e_2$, 我們對兩邊都做融合，看是否能製造出同一個 *foldr*.

例 6.6. 回顧練習 *2.11*: 證明對所有 f, xs, 與 ys, $map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys$. 若把 xs 提出，這相當於證明：

$$map\,f \cdot (+\!\!+ ys) = (+\!\!+ map\,f\ ys) \cdot map\,f \quad .$$

其中 $(+\!\!+ ys)$ 是 *foldr*. 因此我們可使用摺融合與 *foldr-map* 融合:

$$\begin{aligned}
& (+\!\!+ map\,f\ ys) \cdot map\,f \\
= & \quad \{ foldr\text{-}map\ 融合 \} \\
& foldr\ ((:) \cdot f)\ (map\,f\ ys) \\
= & \quad \{ 摺融合 \} \\
& map\,f \cdot (+\!\!+ ys) \quad .
\end{aligned}$$

其中，最後一步的融合條件為

$$\begin{aligned}
& map\,f\ (x : zs) \\
= & \quad \{ map\ 之定義 \} \\
& f\,x : map\,f\ zs \\
= & \quad \{ (\cdot)\ 之定義 \} \\
& ((:) \cdot f)\ x\ (map\,f\ zs) \quad .
\end{aligned}$$

雖然看來複雜，其實是運用符號、展開定義即可證成的性質。

在本節的許多例子中，使用摺融合定理大大簡化了證明 — 幾乎到了只要把式子寫下就快要證完了，「沒什麼可說」的地步。我們再看最後一個例子。

例 6.7. 考慮證明第 2.5節中提及的性質: $sum \cdot concat = sum \cdot map\ sum$. 我們可使用 *foldr-map* 融合定理與摺融合定理:

$$\begin{aligned}
& sum \cdot map\ sum \\
= & foldr\ (+)\ 0 \cdot map\ sum \\
= & \quad \{ foldr\text{-}map\ 融合 \} \\
& foldr\ (\lambda xs\ n \to sum\ xs + n)\ 0 \\
= & \quad \{ 摺融合 \} \\
& sum \cdot foldr\ (+\!\!+)\ [\,] \\
= & sum \cdot concat \quad .
\end{aligned}$$

第二步的 *foldr-map* 融合能成立的原因是 $(+) \cdot sum$ 展開之後確實成為 $(\lambda xs\ n \to sum\ xs + n)$. 這一步也可改用摺融合定理證明，其融合條件為 $sum\ (sum\ xs : ys) = sum\ xs + sum\ ys$, 只需展開定義即可證成。

倒數第二步的摺融合的條件為: $sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$. 這是第 2.5節的證明中必須發明的關鍵性質。我們再一次看到: 使用摺融合定理讓我們只需提供一個證明中最與問題相關的關鍵部分。

習題 6.3 — 請用摺融合證明 *foldr-map* 融合定理 (6.4).

習題 6.4 — 使用 *foldr-map* 融合和摺融合證明 $sum \cdot map\ length = length \cdot concat$.

習題 6.5 — 使用摺融合定理證明對所有 f, $map\ f \cdot concat = concat \cdot map\ (map\ f)$.

習題 6.6 — 給定 $f, g :: a \to \mathsf{List}\ a$ 與 $p :: a \to \mathsf{Bool}$, 試證明：如果 $filter\ p\ (f\ x) = \textbf{if}\ p\ x\ \textbf{then}\ g\ x\ \textbf{else}\ [\,]$, 則 $concat \cdot map\ (filter\ p \cdot f) = concat \cdot map\ g \cdot filter\ p$.

6.2.2 以摺融合生成程式

如前所述，另一種使用摺融合的理由是生成程式：我們希望 $h \cdot foldr\ f\ e$ 能在一個 *foldr* 之中完成。此時我們已知 h, f, 與 e, 希望用融合條件找出適合的步驟函數。

例 6.8. 回顧第 *5.1*節的例子：給定 $sumsq = sum \cdot map\ square$，我們希望找出一個不產生中間串列的版本。由於 $map\ square$ 是一個 *foldr*, 我們嘗試將 *sum* 融合進 $map\ square$ 中，希望找出能滿足 $sumsq = foldr\ g\ e$ 的 g 與 e. 顯然 $e = sumsq\ [\,] = 0$. 為了找出滿足融合條件的步驟函數 g, 我們推算：

$$
\begin{aligned}
& sum\ (square\ x : xs) \\
= & \quad \{\ sum\ 之定義\ \} \\
& square\ x + sum\ xs \\
= & \quad \{\ 提出\ x\ 與\ sum\ xs\ \} \\
& (\lambda x\ y \to square\ x + y)\ x\ (sum\ xs)\ .
\end{aligned}
$$

因此，根據定理 *6.2*, $sumsq = foldr\ (\lambda x\ y \to square\ x + y)\ 0$.

例 6.9. 給定一個整數串列，其中由左到右的數字表示對一個帳戶存款或提款的金額：正數為存款、負數為提款。我們想確定在任何一個時刻帳戶金額不至於變成負數。一個可能做法是：對該串列的每一個前段算總和，我們可得到每個時刻的帳戶金額。接著看看其中最小值是否為負數即可。定義函數 *noOverdraft* 如下：

$$
\begin{aligned}
& noOverdraft :: \mathsf{Int} \to \mathsf{Bool} \\
& noOverdraft = (\geqslant 0) \cdot minimum \cdot map\ sum \cdot inits\ .
\end{aligned}
$$

我們試著導出一個比較快速的版本。

為增進效率，我們試試看能否把 $minimum \cdot map\ sum \cdot inits$ 合併為一個 *foldr*. 回顧：$inits = foldr\ (\lambda x\ xss \to [\,] : map\ (x:)\ xss)\ [\,]$. 我們可以一口氣把 $minimum \cdot map\ sum$ 融合進 *inits*, 也可分兩次進行，先將 $map\ sum \cdot inits$ 融合成一個 *foldr*, 再與 *minimum* 融合。

此處我們嘗試後者，先將 $map\ sum \cdot inits$ 融合。基底值為 $map\ sum\ [[\,]] = [0]$, 而步驟函數 $step_1$ 須滿足的融合條件為 $map\ sum\ ([\,] : map\ (x:)\ xss) = step_1\ x\ (map\ sum\ xss)$. 試計算如下：

$$
\begin{aligned}
& map\ sum\ ([\,]:map\ (x{:})\ xss) \\
= {} & 0:map\ sum\ (map\ (x{:})\ xss) \\
= {} & \quad \{ map\ 融合 \} \\
& 0:map\ (sum \cdot (x{:}))\ xss \\
= {} & \quad \{ sum\ (x:xs) = x + sum\ xs,\ map\ 融合 \} \\
& 0:map\ (x+)\ (map\ sum\ xss)\ .
\end{aligned}
$$

因此我們得到

$$map\ sum \cdot inits = foldr\ (\lambda x\ ys \to 0:map\ (x+)\ ys)\ [0]\ .$$

下一步是將 $minimum$ 融合進 $map\ sum \cdot inits$. 基底值為 $minimum\ [0] = 0$, 而步驟函數 $step_2$ 須滿足 $minimum\ (0:map\ (x+)\ ys) = step_2\ x\ (minimum\ ys)$. 計算如下:

$$
\begin{aligned}
& minimum\ (0:map\ (x+)\ ys) \\
= {} & 0 \downarrow minimum\ (map\ (x+)\ ys) \\
= {} & \quad \{ minimum\ (x+)\ ys = x + minimum\ ys,\ 後述 \} \\
& 0 \downarrow (x + minimum\ ys)\ .
\end{aligned}
$$

最後一步使用的性質 $minimum\ (x+)\ ys = x + minimum\ ys$ 尚待證明, 其關鍵性質是 $(x+)$ 可分配進 $(\downarrow)$ 之中: $x + (y \downarrow z) = (x+y) \downarrow (x+z)$. 總之, 我們得到

$$
\begin{aligned}
& noOverdraft \\
= {} & (\geqslant 0) \cdot minimum \cdot map\ sum \cdot inits \\
= {} & (\geqslant 0) \cdot foldr\ (\lambda x\ y \to 0 \downarrow (x+y))\ 0\ .
\end{aligned}
$$

這是一個只需線性時間的演算法。

我們能否把 $(\geqslant 0)$ 也融入 $foldr$ 之中呢? 要使這個融合成立, 我們得找到滿足 $0 \downarrow (x+y) \geqslant 0 \equiv step_3\ x\ (y \geqslant 0)$ 的 $step_3$. 演算如下:

$$
\begin{aligned}
& 0 \downarrow (x+y) \geqslant 0 \\
\equiv {} & \quad \{ a \downarrow b \geqslant c \equiv a \geqslant c \wedge b \geqslant c \} \\
& 0 \geqslant 0 \wedge x + y \geqslant 0 \\
\equiv {} & x + y \geqslant 0 \\
\equiv {} & \quad \{ 希望找到這樣的\ step_3 \} \\
& step_3\ x\ (y \geqslant 0)\ .
\end{aligned}
$$

然而我們無法找到這樣的 $step_3$ — 僅由 $y \geqslant 0$ 我們無法得知 $x + y \geqslant 0$ 是否成立。我們可說 $(\geqslant 0)$ 丟失了太多資訊, 使得融合無法成立。

也由於同樣的理由, 如果我們最初把問題定義為:

$$noOverdraft = and \cdot map\ (\geqslant 0) \cdot map\ sum \cdot inits\ ,$$

函數 $map\ (\geqslant 0)$ 將無法融合進 $map\ sum \cdot inits$ 之中。

使用摺融合論證兩個式子相等的證明常有如下的形式:

$$
\begin{aligned}
& h_1 \cdot foldr\ f_1\ e_1 \\
= {} & \quad \{ 摺融合定理 \}
\end{aligned}
$$

$$
\begin{aligned}
& foldr\ g\ (h_1\ e_1) \\
= & \quad \{\ 摺融合定理\ \} \\
& h_2 \cdot foldr\ f_2\ e_2\ \ .
\end{aligned}
$$

此時，我們也常需要藉由兩個融合條件之一來發現步驟函數 g 是什麼。

例 6.10. 習題 *2.28* 曾證明 $length \cdot sublists = exp\ 2 \cdot length$. 此處我們用摺融合定理試試看。

考慮等式的左手邊，我們嘗試將 $length \cdot sublists$ 融合為一個 $foldr$. 由於 $sublists = foldr\ (\lambda x\ xss \to xss +\!\!+ map\ (x{:})\ xss)\ [[\,]]$（習題 *6.1(2)*），融合後的 $foldr$ 之基底值為 $length\ [[\,]] = 1$. 為找出步驟函數，我們推算：

$$
\begin{aligned}
& length\ (xss +\!\!+ map\ (x{:})\ xss) \\
= & \ length\ xss + length\ (map\ (x{:})\ xss) \\
= & \quad \{\ length\ (map\ f) = length\ \} \\
& 2 \times length\ xss\ \ ,
\end{aligned}
$$

由此得到步驟函數 $(\lambda x\ n \to 2 \times n)$.

因此該等式可證明如下：

$$
\begin{aligned}
& length \cdot sublists \\
= & \ length \cdot foldr\ (\lambda x\ xss \to xss +\!\!+ map\ (x{:})\ xss)\ [[\,]] \\
= & \quad \{\ 摺融合定理，如上\ \} \\
& foldr\ (\lambda x\ n \to 2 \times n)\ 1 \\
= & \quad \{\ 摺融合定理，如下\ \} \\
& exp\ 2 \cdot foldr\ \mathbf{1}_+\ \mathbf{0} \\
= & \ exp\ 2 \cdot length\ \ .
\end{aligned}
$$

在第二次摺融合中，基底值 $exp\ 2\ \mathbf{0}$ 確實是 1. 融合條件為 $exp\ 2\ (\mathbf{1}_+\ n) = 2 \times exp\ 2\ n$.

習題 6.7 — 回顧例 6.9. 試著將 $map\ (\geqslant 0)$ 融入 $map\ sum \cdot inits$ 中，說說看為何該融合會失敗。

習題 6.8 — 使用摺融合定理證明 $sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$. **提示**：這相當於證明 $sum \cdot (+\!\!+ ys) = (+(sum\ ys)) \cdot sum$.

習題 6.9 — 參考習題 6.2, 使用摺融合定理證明 $length\ (fan\ y\ xs) = \mathbf{1}_+\ (length\ xs)$.

習題 6.10 — 回顧第 5.3.2節中將反轉表示的二進位數字轉為自然數的函數 $decimal :: \mathsf{List\ Bool} \to \mathbb{N}$. 該函數可寫成一個摺：

$$decimal = foldr\ (\lambda c\ n \to \mathbf{if}\ c\ \mathbf{then}\ 1 + 2 \times n\ \mathbf{else}\ 2 \times n)\ 0$$

請使用摺融合將 $exp\ b \cdot decimal$ 表示成單一的摺。

摺融合與尋找歸納定義 回顧：$id :: \text{List } a \to \text{List } a$ 可以寫成一個 $foldr - id = foldr\ (:)\ [\,]$. 而任何函數 $f :: \text{List } a \to b$ 都等於 $f \cdot id$. 如果我們將 f 與 id 融合，會發生什麼事呢？首先我們需要找出基底值 $f\ [\,]$ 是什麼。接著我們要找到滿足 $f\ (x : xs) = step\ x\ (f\ xs)$ 的步驟函數 $step$. 但這其實就是使用展開-收回轉換尋找 f 的歸納定義！只是此處要求的歸納定義比較嚴格：在 $f\ xs$ 之外不能使用 xs.

確實，第 5.1與 5.3節中許多尋找歸納定義的演算都可以視為使用摺融合生成程式的例子。以第 5.3.1節的 $poly$ 為例。找出其歸納定義的過程可以視為摺融合：

$$\begin{aligned}
&\quad poly\ x \\
&= poly\ x \cdot id \\
&= \quad \{\ id = foldr\ (:)\ [\,]\ \} \\
&\quad poly\ x \cdot foldr\ (:)\ [\,] \\
&= \quad \{\ \text{摺融合定理}\ \} \\
&\quad foldr\ step\ (poly\ x\ [\,])\ \ .
\end{aligned}$$

其中基底值 $poly\ x\ [\,] = 0$. 而函數 $step$ 須滿足融合條件 $poly\ x\ (a : as) = step\ a\ (poly\ x\ as)$。尋找 $step$ 的過程和第 114頁的計算完全相同。我們會得到

$$poly\ x\ (a : as) = a + (poly\ x\ as) \times x\ \ .$$

到此為止我們便找到了 $poly\ x$ 的歸納定義。也可以說，我們已得知 $poly\ x = foldr\ (\lambda a\ b \to a + b \times x)\ 0$.

6.2.3 掃描

本節將介紹一個本書首次提及，初見時較難理解，但在許多演算法中扮演重要角色的組件函數：掃描（*scan*）。

如我們所知，函數 $sum :: \text{List Int} \to \text{Int}$ 計算一個串列的總和。如果我們想計算一個串列由右到左的累計和，例如當給定串列 $[3,7,2,4]$，我們希望得到 $[16,13,6,4,0]$（其中 $6 = 2+4$, $13 = 7+2+4$，$16 = 3+7+2+4$, 而 0 是空串列的和），該怎麼做呢？

在第 2.6.3節中，我們曾提及計算一個串列所有後段 (*suffixes*) 的函數 $tails :: \text{List } a \to \text{List (List } a)$。例如，$tails\ [3,7,2,4]$ 將得到 $[[3,7,2,4], [7,2,4], [2,4], [4], [\,]]$。對串列的每一個後段算總和，我們便得到累計和 $[16,13,6,4,0]$ 了：

$$\begin{aligned}
&runsum :: \text{List Int} \to \text{List Int} \\
&runsum = map\ sum \cdot tails\ \ .
\end{aligned}$$

由於使用多個 sum 函數走訪每個後段，如此定義出的 $runsum$ 將是一個執行時間為 $O(n^2)$ 的函數。但讀者想必已覺得可不用如此費事：我們應該可以在由右到左走訪串列的過程中記住目前為止的和，避免重算 sum。這該怎麼做呢？

回想：sum 可寫成一個摺。因此我們可稍微推廣一下，定義函數 $scanr$ 如下：

$$\begin{aligned}
&scanr :: (a \to b \to b) \to b \to \text{List } a \to \text{List } b \\
&scanr\ f\ e = map\ (foldr\ f\ e) \cdot tails\ \ .
\end{aligned}$$

給定一個串列 xs, $scanr\ f\ e$ 先算出 xs 的所有後段，然後對每一個後段都做 $foldr\ f\ e$. 前述的 $runsum$ 其實是 $scanr$ 的特例：$runsum = scanr\ (+)\ 0$.

如果把上述的 $scanr$ 定義當作演算法，處理長度為 n 的串列時呼叫 f 的次數為 $O(n^2)$. 我們找找看是否有比較快的演算法。

第 6.1.1 節中提及 $tails$ 是一個 $foldr$:

$$tails = foldr\ (\lambda x\ xss \to (x : head\ xss) : xss)\ [[\,]]\ ,$$

也許我們可試著把 $map\ (foldr\ f\ e)$ 融合入 $tails$ 中，看看是否能找出一個較有效率的 $scanr$ 定義。其融合條件如下：

$$\begin{aligned}
&map\ (foldr\ f\ e)\ ((x : head\ xss) : xss)\\
=\ &\{\ map\ 之定義\ \}\\
&foldr\ f\ e\ (x : head\ xss) : map\ (foldr\ f\ e)\ xss\\
=\ &\{\ foldr\ 之定義\ \}\\
&f\ x\ (foldr\ f\ e\ (head\ xss)) : map\ (foldr\ f\ e)\ xss\\
=\ &\{\ g\ (head\ ys) = head\ (map\ g\ ys);\ 令\ g := foldr\ f\ e,\ ys := xss\ \}\\
&f\ x\ (head\ (map\ (foldr\ f\ e)\ xss)) : map\ (foldr\ f\ e)\ xss\\
=\ &\{\ 取出\ map\ (foldr\ f\ e)\ xss\ \}\\
&\mathbf{let}\ ys = map\ (foldr\ f\ e)\ xss\\
&\mathbf{in}\ f\ x\ (head\ ys) : ys\ \ .
\end{aligned}$$

於是我們推導出了 $scanr$ 的另一個定義：

引理 6.11 (掃描引理). 對所有 f, e,

$$scanr\ f\ e = foldr\ (\lambda x\ ys \to f\ x\ (head\ ys) : ys)\ [e]\ \ .$$

若將 $foldr$ 的定義展開，函數 $scanr$ 可寫成下列的歸納形式，也許比較容易理解：

$$\begin{aligned}
&scanr\ f\ e\ [\,] &&= [e]\\
&scanr\ f\ e\ (x : xs) &&= f\ x\ (head\ ys) : ys\ \ ,\\
&\quad \mathbf{where}\ ys = scanr\ f\ e\ xs\ \ .
\end{aligned}$$

在第 6.1.1 節中，使得 $tails$ 能寫成一個 $foldr$ 的重要性質是 $head\ (tails\ xs) = xs$ – $tails\ xs$ 的第一個元素就是 xs 本身。使用 $tails$ 定義的 $scanr$ 自然繼承了相關的性質：$scanr\ f\ e\ xs$ 的第一個元素就是 $foldr\ f\ e\ xs$, 因此可直接用 $head$ 取出，不需每次重新計算。用本節開頭的例子說明，以下我們令 $scr = (\lambda x\ ys \to x + head\ ys : ys)$:

$$\begin{aligned}
&scanr\ (+)\ 0\ [3,7,2,4]\\
&= scr\ 3\ (scr\ 7\ (scr\ 2\ (scr\ 4\ [0])))\\
&= scr\ 3\ (scr\ 7\ (scr\ 2\ [4+0,0]))\\
&= scr\ 3\ (scr\ 7\ [2+4+0,4+0,0])\\
&= scr\ 3\ [7+2+4+0,2+4+0,4+0,0]\\
&= [3+7+2+4+0,7+2+4+0,2+4+0,4+0,0]\\
&= [16,13,6,4,0]\ \ ,
\end{aligned}$$

其中每個 scr 都可直接使用之前累積計算的結果，不用從頭加起。

6.2.4 香蕉船定理

第 5.5.4節簡短地提到一個例子：令 $sumlen = \langle sum, length \rangle$,[1] 直接執行的話，$sum$ 與 $length$ 將各自走訪輸入串列一次，但我們可推導出 $sumlen$ 的歸納定義，得到一個只走訪串列一次的版本。

上述例子還可以更通用一些。考慮 $\langle foldr\, f_1\, e_1, foldr\, f_2\, e_2 \rangle$ — 這個算式拿一個串列當輸入，兩個 $foldr$ 分別將串列走訪一次，兩個結果分別存放在序對中。我們有可能將它變成一個摺（因此只走訪串列一次）嗎？下述的「香蕉船定理 (banana-split theorem)」告訴我們：含兩個摺的分裂，可以寫成一個摺。

定理 6.12 (香蕉船定理). 給定 $f_1 :: a \to b \to b, e_1 :: b, f_2 :: a \to c \to c, e_2 :: c$, 下述等式成立：

$$\langle foldr\, f_1\, e_1, foldr\, f_2\, e_2 \rangle = foldr\, g\, (e_1, e_2) \ ,$$

其中 $g\, x\, (y, z) = (f_1\, x\, y, f_2\, x\, z)$.

分裂運算元 $\langle \cdot, \cdot \rangle$ 的英文稱呼是 'split", 在程式推導圈內有時會用一套稱作「香蕉括號 (banana brackets)」的符號表示摺，兩者合起來便是 'banana-split" — 甜點「香蕉船」的英文名稱。

定理 6.12相當於把兩個處理同一份資料的迴圈合併成一個。但如同第 5.5.4節提及，這並不保證效率會比較好。我們會使用定理 6.12的原因可能是如果確定某函數是摺，我們能做更多後續處理（例如，使用掃描定理或其他只對摺成立的性質）。

> **習題 6.11** — 證明香蕉船定理。你可以在輸入串列上做歸納證明，也可以利用 $\langle foldr\, f_1\, e_1, foldr\, f_2\, e_2 \rangle = \langle foldr\, f_1\, e_1, foldr\, f_2\, e_2 \rangle \cdot id = \langle foldr\, f_1\, e_1, foldr\, f_2\, e_2 \rangle \cdot foldr\, (:)\, [\,]$ 的特性，使用摺融合定理。

「組對」常可視為分裂與 id 的融合。例如，在第 5.5.1節的陡串列問題中，我們定義 $steepsum\, xs = (steep\, xs, sum\, xs)$，並試著推導其歸納定義。該過程也可視為將 $\langle steep, sum \rangle$ 與 id 融合：

$$\begin{aligned}
& \langle steep, sum \rangle \\
= \quad & \{ f \cdot id = f \} \\
& \langle steep, sum \rangle \cdot id \\
= \quad & \{ id = foldr\, (:)\, id \} \\
& \langle steep, sum \rangle \cdot foldr\, (:)\, [\,] \\
= \quad & \{ \text{摺融合} \} \\
& foldr\, (\lambda x\, (b, s) \to (x > s \wedge b, x + s))\, (\mathsf{True}, 0) \ .
\end{aligned}$$

其融合條件的證明與第 5.5.1節中幾乎相同。

[1] 分裂運算元 $\langle \cdot, \cdot \rangle$ 的定義請參照第 1.6.3節，36頁。

6.2.5 累積參數與摺融合

第 5.6節介紹的「累積參數」技巧也常可視為高階函數與摺的融合。以第 5.6.1節的經典例子 — 串列反轉為例。函數 $reverse :: \mathsf{List}\ a \to \mathsf{List}\ a$ 是一個 *foldr*:

$$reverse = foldr\ (\lambda x\ xs \to xs +\!\!+ [x])\ [] \ .$$

為增進其效率，我們創造了函數 *revcat*，其定義為：

$$\begin{aligned} &revcat :: \mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a \\ &revcat\ xs\ ys = reverse\ xs +\!\!+ ys \ . \end{aligned}$$

但如果把參數都移除，上述定義其實等同於：

$$revcat = (+\!\!+) \cdot reverse \ .$$

推導 *revcat* 的歸納定義就是計算 $(+\!\!+)$ 與 *reverse* 的融合！

為了導出一個較快的 *revcat* 實作，我們嘗試把 $(+\!\!+) \cdot reverse$ 融合為一個 *foldr*. 其推導大綱如下：

$$\begin{aligned} &(+\!\!+) \cdot reverse \\ &= (+\!\!+) \cdot foldr\ (\lambda x\ xs \to xs +\!\!+ [x])\ [] \\ &= \quad \{\ \text{摺融合，試著計算出 } base \text{ 與 } step\ \} \\ &\quad foldr\ step\ base \ . \end{aligned}$$

我們可暫停一下，看看這個式子的型別。函數 $(+\!\!+)$ 的型別為 $\mathsf{List}\ a \to (\mathsf{List}\ a \to \mathsf{List}\ a)$, $(+\!\!+) \cdot reverse$ 與 *foldr step base* 的型別也相同。如果摺融合成功，我們會得到的是一個傳回函數的 *foldr* — 輸入為 $\mathsf{List}\ a$, 輸出為 $\mathsf{List}\ a \to \mathsf{List}\ a$. 其中 *base* 的型別為 $\mathsf{List}\ a \to \mathsf{List}\ a$, 而 *step* 的型別將是 $a \to (\mathsf{List}\ a \to \mathsf{List}\ a) \to (\mathsf{List}\ a \to \mathsf{List}\ a)$ — *step x* 將一個函數轉成另一個函數。

根據摺融合定理，基底值 *base* 是

$$\begin{aligned} (+\!\!+)\ [] &= (\lambda xs \to (+\!\!+)\ []\ xs) \\ &= (\lambda xs \to [] +\!\!+ xs) \\ &= (\lambda xs \to xs) \\ &= id \ . \end{aligned}$$

步驟函數 *step* 須滿足的融合條件如下

$$(+\!\!+)\ ((\lambda x\ xs \to xs +\!\!+ [x])\ x\ xs) = step\ x\ ((+\!\!+)\ xs) \ .$$

簡單地化簡等號左手邊，我們得到：

$$(+\!\!+)\ (xs +\!\!+ [x]) = step\ x\ ((+\!\!+)\ xs) \ .$$

這個式子無法再規約，因為 $(+\!\!+)$ 還需要一個參數。因此我們根據外延相等（定義 1.8），在等號兩邊各補一個參數 *ys*:

$$(+\!\!+)\ (xs +\!\!+ [x])\ ys = step\ x\ ((+\!\!+)\ xs)\ ys \ .$$

為找出 *step*, 演算如下：

$$
\begin{aligned}
& (+\!\!+)\ (xs +\!\!+ [x])\ ys \\
= & \ (xs +\!\!+ [x]) +\!\!+ ys \\
= & \quad \{\ (+\!\!+) \text{ 之結合律}\ \} \\
& \ xs +\!\!+ ([x] +\!\!+ ys) \\
= & \quad \{\ (\cdot) \text{ 之定義}\ \} \\
& \ (((+\!\!+)\ xs) \cdot (x:))\ ys \\
= & \quad \{\ \text{將 } x, ((+\!\!+)\ xs), \text{與 } ys \text{ 提出}\ \} \\
& \ (\lambda x\, f \to f \cdot (x:))\ x\ ((+\!\!+)\ xs)\ ys \quad .
\end{aligned}
$$

根據外延相等，我們已證明

$$(+\!\!+)\ (xs +\!\!+ [x]) = (\lambda x\, f \to f \cdot (x:))\ x\ ((+\!\!+)\ xs) \quad .$$

因此 $step = (\lambda x\, f \to f \cdot (x:))$, 而 *revcat* 可寫成如下的摺：

$$revcat = foldr\ (\lambda x\, f \to f \cdot (x:))\ id \quad .$$

例如 $revcat\ \texttt{"abc"} = id \cdot (\texttt{'c'}:) \cdot (\texttt{'b'}:) \cdot (\texttt{'a'}:)$, 而 $revcat\ \texttt{"abc"}\ ys = \texttt{'c'} : (\texttt{'b'} : (\texttt{'a'} : ys))$.

6.2.6 引入脈絡

第 6.2節開頭(第 144頁)曾提及：使用摺融合定理將 $h \cdot foldr\, f\, e$ 融合成 $foldr\, g\, (h\, e)$ 時，融合條件 $h\ (f\ x\ y) = g\ x\ (h\ y)$ 並不需對所有 y 都成立，而只需對在 $foldr\, f\, e$ 的值域內的 y 成立即可。截至目前為止我們看了不少摺融合的例子，但我們所證明的融合條件，均是較寬鬆、對所有 y 都成立的。我們還沒看過只對特定 y 成立（因此可能較難證明的）融合條件。

知道「y 在 $foldr\, f\, e$ 的值域內」，意味著證明融合條件時，我們可以假設 y 滿足所有 $foldr\, f\, e$ 的傳回值該滿足的性質。習慣上，「使用這些性質」被稱作「引入脈絡」(*bringing in the context*) — 意指做證明時知道 y 不是任意的值，而是由 $foldr\, f\, e$ 產生（有特定脈絡）的。此處我們看一個引入脈絡的簡單例子。

例 6.13. 回顧 $sumsq = sum \cdot map\ square$. 我們在例 6.8中曾導出 $sumsq = foldr\ (\lambda x\, y \to square\ x + y)\ 0$. 現在考慮下述函數：

$$psuc\ n = \textbf{if } n \geqslant 0 \textbf{ then } n+1 \textbf{ else } 0 \quad .$$

如果參數是非負整數，*psuc* 將它加一，否則傳回 0. 我們能將 $psuc \cdot sumsq$ 融合為一個 *foldr* 嗎?

直覺看來，由於 *sumsq* 一定傳回非負整數，*psuc* 只是將其結果加一。但如果使用摺融合，需要的融合條件是 $psuc\ (square\ x + y) = step\ x\ (psuc\ y)$. 我們找不到一個對任意 x 與 y 都能使該條件成立的 *step*.

此時我們需要引入脈絡：由於 y 是 *sumsq* 的結果，必定是個非負整數。我們可證明對於非負整數 m 與 n,

$$psuc\ (m+n) = m + psuc\ n \quad .$$

由於 $square\ x$ 與 y 都是非負整數，我們有

$$psuc\ (square\ x + y) = square\ x + psuc\ y \quad .$$

因此我們可選 $step\ x\ y = square\ x + y$ — 與 $sumsq$ 的步驟函數相同。至於基底值則是 $psuc\ 0 = 1$. 因此，

$$psuc \cdot sumsq = foldr\ (\lambda x\ y \to square\ x + y)\ 1 \quad .$$

例 6.13是個刻意設計的、簡單的例子。在第 TODO 節中，我們會看到其他需要引入脈絡的演算法。

6.3 左摺、串列同構、與 Paramorphism

為了完整性，本節介紹與串列的摺相關的另外幾個組件函數。初次閱讀時可跳過本節。

6.3.1 左摺

除了 $foldr$, Haskell 的標準函式庫有另一個函數 $foldl$, 名字中的字母 l 為「左」之意：$foldr$ 將串列中的元素往右括，$foldl$ 則往左括，例如，

$$foldl\ (\triangleright)\ e\ [x_0, x_1, x_2, x_3] = (((e \triangleright x_0) \triangleright x_1) \triangleright x_2) \triangleright x_3 \quad .$$

有了 $foldl$, 串列反轉可直接定義為 $reverse = foldl\ (\lambda xs\ x \to x : xs)\ [\,]$.

函數 $foldl$ 可在輸入串列的長度上歸納定義如下：

$$\begin{aligned} &foldl :: (b \to a \to b) \to b \to \mathsf{List}\ a \to b \\ &foldl\ (\triangleright)\ e\ [\,] \qquad\quad = e \\ &foldl\ (\triangleright)\ e\ (xs +\!\!+ [z]) = foldl\ (\triangleright)\ e\ xs \triangleright z \quad . \end{aligned}$$

由於運算元 $(\triangleright)$ 右邊的參數才是目前的元素，其型別為 $b \to a \to b$. 由此出發，我們不難導出下述將串列從左邊開始拆解的定義：

$$\begin{aligned} &foldl\ (\triangleright)\ e\ [\,] \quad\ = e \\ &foldl\ (\triangleright)\ e\ (x : xs) = foldl\ (\triangleright)\ (e \triangleright x)\ xs \quad . \end{aligned}$$

我們可注意到這是一個尾遞迴定義，因此，如果將 sum, $prod$ 等函數定義為 $foldl\ (+)\ 0$, $foldl\ (\times)\ 1$ 等等，執行時可不佔用堆疊的空間。由於這個特性，對一些將 Haskell 用於著重效率的應用的人們來說，$foldl$ 才是他們較常使用的「摺」。[2]

關於 $foldl$ 與 $foldr$ 的關係，Bird [1998] 提及了三個定理。首先，

[2] 為了效率因素更常被使用的可能是另一個嚴格 (strict) 版的函數 $foldl'$, 該函數在遞迴呼叫前會先將 $e \triangleright x$ 規約成正規式，避免尚待計算的算式被累積著。

定理 6.14 (第一對偶定理 (The First Duality Theorem)). 如果 $(\oplus) :: a \to a \to a$ 滿足遞移律，以 $e :: a$ 為單位元素，則 $foldr\ (\oplus)\ e\ xs = foldl\ (\oplus)\ e\ xs$.

提醒一下：該定理及本節的其他定理中，xs 需為歸納定義、有限長度的串列。遞移律意指 $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ — 由於此項要求，$(\oplus)$ 的型別得是 $a \to a \to a$. 由於 $(+)$, $(\times)$, $(\downarrow)$, $(\uparrow)$ 等運算元都有遞移律，sum, $prod$, $maixmum$, $minimum$ 等函數都可用 $foldl$ 定義。事實上，他們在標準 Haskell Report 中的定義便是如此。

定理 6.14其實是下述定理的特例。函數 $foldl$ 有個尾遞迴定義，其中參數 e 的角色就像是個累積參數。第 5.6.2節中曾提及，尾遞迴函數時常和遞移律有關。函數 $foldl$ 是否也牽涉了某種遞移律呢？

定理 6.15 (第二對偶定理 (The Second Duality Theorem)). 如果二元運算元 $(\lhd) :: a \to b \to b$ 及 $(\rhd) :: b \to a \to b$ 滿足 $x \lhd (y \rhd z) = (x \lhd y) \rhd z$ 以及 $x \lhd e = e \rhd x$，則 $foldr\ (\lhd)\ e\ xs = foldl\ (\rhd)\ e\ xs$.

定理 6.15 中的 $x \lhd (y \rhd z) = (x \lhd y) \rhd z$ 可視為一種擴充到兩個運算元的遞移律。定理 6.14 則是 $(\lhd) = (\rhd)$ 的特殊情況。

定理 6.15 的證明頗值得研究：

Proof. 我們在 xs 上做歸納。基底狀況 $xs := []$ 中，等號兩邊都歸約為 e. 至於 $xs := x : xs$ 的情況，我們把兩邊都化簡並使用歸納假設，推論如下：

$$
\begin{aligned}
& foldr\ (\lhd)\ e\ (x : xs) \\
= \quad & \{ foldr \text{ 之定義} \} \\
& x \lhd foldr\ (\lhd)\ e\ xs \\
= \quad & \{ \text{歸納假設} \} \\
& x \lhd foldl\ (\rhd)\ e\ xs \\
= \quad & \{ \text{論證如後述} \} \\
& foldl\ (\rhd)\ (x \lhd e)\ xs \\
= \quad & \{ \text{假設：} x \lhd e = e \rhd x \} \\
& foldl\ (\rhd)\ (e \rhd x)\ xs \\
= \quad & \{ foldl \text{ 之定義} \} \\
& foldl\ (\rhd)\ e\ (x : xs) \quad .
\end{aligned}
$$

在等式推論中段，我們希望 $x \lhd foldl\ (\rhd)\ e\ xs = foldl\ (\rhd)\ (e \rhd x)\ xs$. 這會需要另一個歸納證明。但我們若直接證明此等式，到中途便會無法進行，且會發現我們需要稍做一下推廣，改證明一個較強的性質：

$$x \lhd foldl\ (\rhd)\ y\ xs = foldl\ (\rhd)\ (x \lhd y)\ xs \quad . \tag{6.1}$$

兩者的差別是後者不針對特定的值 e, 而是任何的 y. 這是一個「較強的性質反而比較好證明」的例子。

等式(6.1)的證明如此進行：在 $xs := [\,]$ 的情況中，等號兩側都規約為 $x \lhd y$. 歸納情況 $xs := z : xs$ 證明如下：

$$
\begin{aligned}
& x \lhd \mathit{foldl}\ (\rhd)\ y\ (z : xs) \\
= \quad & \{\ \mathit{foldl}\ \text{之定義}\ \} \\
& x \lhd \mathit{foldl}\ (\rhd)\ (y \rhd z)\ xs \\
= \quad & \{\ \text{歸納假設}\ \} \\
& \mathit{foldl}\ (\rhd)\ (x \lhd (y \rhd z))\ xs \\
= \quad & \{\ \text{假設：} x \lhd (y \rhd z) = (x \lhd y) \rhd z\ \} \\
& \mathit{foldl}\ (\rhd)\ ((x \lhd y) \rhd z)\ xs \\
= \quad & \{\ \mathit{foldl}\ \text{之定義}\ \} \\
& \mathit{foldl}\ (\rhd)\ (x \lhd y)\ (z : xs)\ .
\end{aligned}
$$

□

最後一個對偶定理則確認了我們的一個直覺理解：將一個串列反轉做 *foldl*, 便相當於 *foldr*:

定理 6.16 (第三對偶定理 (The Third Duality Theorem)). 對所有 $f :: a \to b \to b$, $e :: b$, 以及 $xs :: \mathrm{List}\ a$, $\mathit{foldr}\ f\ e\ xs = \mathit{foldl}\ (\mathit{flip}\ f)\ e\ (\mathit{reverse}\ xs)$. 其中 $\mathit{flip}\ f\ x\ y = f\ y\ x$.

6.3.2 串列同構

考慮函數 $h :: \mathrm{List}\ a \to b$. 如果存在 $e :: b$, $f :: a \to b$, 和 $(\odot) :: b \to b \to b$ 使得 h 滿足下列等式：

$$
\begin{aligned}
h\ [\,] &= e \\
h\ [x] &= f\ x \\
h\ (xs +\!\!+ ys) &= h\ xs \odot h\ ys\ ,
\end{aligned}
$$

我們便說 h 是一個串列同構 (*list homomorphism*), 記為 $h = \mathit{hom}\ (\odot)\ f\ e$. 注意第三個等式蘊含 $(\odot)$（至少在 h 的值域內）須滿足結合律：$h\ xs \odot (h\ ys \odot h\ zs) = h\ (xs +\!\!+ (ys +\!\!+ zs)) = h\ ((xs +\!\!+ ys) +\!\!+ zs) = (h\ xs \odot h\ ys) \odot h\ zs$.

串列同構有平行計算的可能性：計算 $\mathit{hom}\ (\odot)\ f\ e\ xs$ 時，如果 xs 有不只一個元素，我們可將 xs 由中間任意截成兩段，分別給不同的處理器計算，再將結果用 $(\odot)$ 結合。

顯然，$\mathit{hom\ dot}\ f\ e$ 可以寫成 *foldr*, 也能寫成 *foldl*:

$$
\begin{aligned}
\mathit{hom}\ (\odot)\ f\ e &= \mathit{foldr}\ (\lambda x\ y \to f\ x \odot y)\ e \\
&= \mathit{foldl}\ (\lambda y\ x \to y \odot f\ x)\ e\ .
\end{aligned}
$$

下述定理則告訴我們反過來也成立：如果 h 同時可寫成 *foldr* 及 *foldl*, 則 h 是一個串列同構：

定理 6.17 (第三同構定理 (The Third Homomorphism Theorem)). 考慮 $h :: \mathrm{List}\ a \to b$。如果存在 $e :: b$, $(\lhd) :: a \to b \to b$, 及 $(\rhd) :: b \to a \to b$ 使得 $h = \mathit{foldr}\ (\lhd)\ e = \mathit{foldl}\ (\rhd)\ e$, 則存在 $(\odot) :: b \to b \to b$ 使得 $h = \mathit{hom}\ (\odot)\ f\ e$（其中 $f\ x = x \lhd e = e \rhd x$）.

6.3.3 Paramorphism 與本原遞迴

如果我們把 $foldr$ 的限制放寬，允許 xs 出現在遞迴呼叫之外，得到的模式稱為 *paramorphism*. 串列版的 paramorphism 可定義如下：

$$\begin{aligned} &para :: (a \to \mathsf{List}\ a \to b \to b) \to b \to \mathsf{List}\ a \to b \\ &para\ f\ e\ [\,] \quad\ = e \\ &para\ f\ e\ (x : xs) = f\ x\ xs\ (para\ f\ e\ xs)\ . \end{aligned}$$

$$para\ f\ e = fst \cdot foldr\ (\lambda x\ (y, xs) \to (f\ x\ xs\ y, x : xs))\ (e, [\,])\ .$$

6.4 自然數的摺

自然數可以視為有兩個建構元 $\mathbf{0}$ 和 $\mathbf{1}_+$ 的歸納資料結構。我們也可為自然數定義一個摺：

$$\begin{aligned} &foldN :: (a \to a) \to a \to \mathbb{N} \to a \\ &foldN\ f\ e\ \mathbf{0} \quad\ = e \\ &foldN\ f\ e\ (\mathbf{1}_+\ n) = f\ (foldN\ f\ e\ n)\ . \end{aligned}$$

和串列的情況類似，函數 $foldN\ f\ e$ 拿一個自然數，將其中的 $\mathbf{0}$ 代換成 e, $\mathbf{1}_+$ 代換成 f. 留意型別：$\mathbf{0}$ 與 $\mathbf{1}_+$ 的型別分別是 $\mathbb{N}$ 與 $\mathbb{N} \to \mathbb{N}$, $foldN$ 要將一個型別為 $\mathbb{N}$ 的值轉變為型別為 a 的值, 因此 e 的型別為 a, 而 f 的型別為 $a \to a$. 為方便稱呼，我們也將 e 與 f 分別稱為基底值與步驟函數。

許多我們定義過的、自然數上的函數都可寫成自然數的摺：

- $exp\ b = foldN\ (b\times)\ 1$,
- $(+n) = foldN\ (\mathbf{1}_+)\ n$,
- $(\times n) = foldN\ (n+)\ 0$.

自然數上的 $id :: \mathbb{N} \to \mathbb{N}$ 也可寫成摺：$id = foldN\ (\mathbf{1}_+)\ 0$.

自然數的摺也有一個融合定理：

定理 6.18 (摺融合定理 (自然數版)). 給定 $f :: a \to a$, $e :: a$, $h :: a \to b$. 如果對所有在 $foldN\ f\ e$ 值域中的 $x :: a$, 融合條件 $h\ (f\ x) = g\ (h\ x)$ 成立, 則

$$h \cdot foldN\ f\ e = foldN\ g\ (h\ e)\ .$$

例 6.19. 判斷一個自然數是否為偶數的述語 $even :: \mathbb{N} \to \mathsf{Bool}$ 可寫成一個摺：

$$even = foldN\ not\ \mathsf{True}\ .$$

函數 $even \cdot (+n)$ 判斷一個數值加上 n 之後是否為偶數。由於 $(+n) = foldN\ (\mathbf{1}_+)\ n$, 我們可以嘗試把 $even$ 融入 $(+n)$, 變成一個摺。根據定理 *6.18*, 基底值為 $even\ n$; 而由於 $even\ (\mathbf{1}_+\ n) = not\ (even\ n)$, 步驟函數為 not. 因此：

$$even \cdot (+n) = foldN\ not\ (even\ n)\ .$$

習題 6.12 — 回顧第 2.11節中提到的費氏數定義：

$$\begin{aligned} &fib\,0 &&= 0 \\ &fib\,1 &&= 1 \\ &fib\,(2+n) &&= fib\,(1+n) + fib\,n \ . \end{aligned}$$

若直接將上述定義當作演算法，我們得做許多重複的計算。定義 $fib2\ n = (fib\,(1+n), fib\,n)$. 請將 $fib2$ 融合進 $id :: \mathbb{N} \to \mathbb{N}$, 以便得到一個遞迴呼叫次數為 $O(n)$ 的演算法。

6.5 其他資料結構

既然串列與自然數都有摺，其他的資料結構也可以有。

二元樹 回顧我們提及的兩種常見二元樹：

$$\begin{aligned} &\textbf{data}\ \mathsf{ITree}\ a &&= \mathsf{Null} \mid \mathsf{Node}\ a\ (\mathsf{ITree}\ a)\ (\mathsf{ITree}\ a) \ , \\ &\textbf{data}\ \mathsf{ETree}\ a &&= \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a) \ . \end{aligned}$$

其中 ITree 的摺可定義如下：

$$\begin{aligned} &foldIT :: (a \to b \to b \to b) \to b \to \mathsf{ITree}\ a \to b \\ &foldIT\ f\ e\ \mathsf{Null} &&= e \\ &foldIT\ f\ e\ (\mathsf{Node}\ x\ t\ u) &&= f\ x\ (foldIT\ f\ e\ t)\ (foldIT\ f\ e\ u) \ . \end{aligned}$$

內標記二元樹 ITree 的兩個建構元之型別分別為 $\mathsf{Null} :: \mathsf{ITree}\ a$ 與 $\mathsf{Node} :: a \to \mathsf{ITree}\ a \to \mathsf{ITree}\ a \to \mathsf{ITree}\ a$. 和串列的摺一樣，內標記二元樹的摺將一個 $\mathsf{ITree}\ a$ 轉成一個型別為 b 的值 — 藉由將 Null 代換為基底值 $e :: b$, 以及將 Node 代換為步驟函數 $f :: a \to b \to b \to b$.

外標記二元樹的摺則可定義如下：

$$\begin{aligned} &foldET :: (b \to b \to b) \to (a \to b) \to \mathsf{ETree}\ a \to b \\ &foldET\ f\ k\ (\mathsf{Tip}\ x) &&= k\ x \\ &foldET\ f\ k\ (\mathsf{Bin}\ t\ u) &&= f\ (foldET\ f\ k\ t)\ (foldET\ f\ k\ u) \ . \end{aligned}$$

型別 ETree 的建構元分別為 $\mathsf{Tip} :: a \to \mathsf{ETree}\ a$ 和 $\mathsf{Bin} :: \mathsf{ETree}\ a \to \mathsf{ETree}\ a \to \mathsf{ETree}\ a$. 由於 Tip 是一個由 a 到 $\mathsf{ETree}\ a$ 的函數，取代它的得是一個型別為 $a \to b$ 的基底函數. 取代 Bin 的步驟函數之型別則為 $b \to b \to b$. 有了這兩者，我們便可將 $\mathsf{ETree}\ a$ 轉換為 b.

例如，第 2.8節中曾提到幾個定義在樹之上的函數：$tags$ 傳回一個 ITree 的所有標記；$size$ 傳回其大小；$minE$ 傳回一個 ETree 的最小元素，$mapE\ f$ 對樹中的每個標記做 f. 它們都可用摺定義：

$$\begin{aligned} &tags &&= foldIT\ (\lambda x\ xs\ ys \to xs +\!\!+ [x] +\!\!+ ys)\ [\,] \ , \\ &size &&= foldIT\ (\lambda x\ m\ n \to 1 + m + n)\ 0 \ , \end{aligned}$$

$$\begin{aligned} minE &= foldET\ (\downarrow)\ id\ , \\ mapE\ f &= foldET\ \mathsf{Bin}\ f\ . \end{aligned}$$

兩個二元樹的摺也有它們的融合定理：

定理 6.20 (摺融合定理 (ITree版)). 給定 $f :: a \to b \to b \to b$, $e :: b$, $h :: b \to c$, 與 $g :: a \to c \to c \to c$. 如果融合條件 $h\ (f\ x\ y\ z) = g\ x\ (h\ y)\ (h\ z)$ 對任何 $x :: a$ 與在 $foldIT\ f\ e$ 值域中的 $y, z :: b$ 成立，我們有 $h \cdot foldIT\ f\ e = foldIT\ g\ (h\ e)$.

定理 6.21 (摺融合定理 (ETree版)). 給定 $f :: b \to b \to b$, $k :: a \to b$, $h :: b \to c$, 與 $g :: c \to c \to c$. 如果融合條件 $h\ (f\ x\ y) = g\ (h\ x)\ (h\ y)$ 對任何在 $foldET\ f\ k$ 值域中的 $x, y :: b$ 成立，我們有 $h \cdot foldET\ f\ k = foldET\ g\ (h \cdot k)$.

兩個定理的融合條件都依循著與串列版相同的原則：當 h 與步驟函數碰在一起，融合條件讓我們將 h 往裡推。兩個定理都可用單純的歸納法證明。

習題 6.13 — 以摺融合定理證明 $length\ (tags\ t) = size\ t$.

習題 6.14 — 串列有 *foldr-map* 融合定理 (6.4)，ETree 也有類似的 *foldET-mapE* 融合定理。請寫出該定理並證明之。

習題 6.15 — 函數 $mapI :: (a \to b) \to \mathsf{ITree}\ a \to \mathsf{ITree}\ b$ 將一個 $a \to b$ 的函數作用在 ITree 的每一個標記上。請用 *foldIT* 定義 *mapI*, 並寫下 *foldIT* 與 *mapI* 的融合定理並證明之。

習題 6.16 — 以摺融合定理證明 $minE\ (mapE\ (x+)\ t) = x + minE\ t$.

習題 6.17 — 將 $(+\!\!+) \cdot tags$ 融合，以便推導出一個在線性時間內收集 ITree 內所有標籤的演算法。

非空串列 第 2.10節中曾提及我們可把至少有一個元素的串列想像成一個資料結構：$\mathbf{data}\ \mathsf{List}^+\ a = [a] \mid a : \mathsf{List}^+\ a$. 此種串列的摺可定義為：

$$\begin{aligned} &foldrn :: (a \to b \to b) \to (a \to b) \to \mathsf{List}^+\ a \to b \\ &foldrn\ f\ k\ [x] \quad\ = k\ x \\ &foldrn\ f\ k\ (x : xs) = f\ x\ (foldrn\ f\ k\ xs)\ . \end{aligned}$$

例 6.22. 例 2.13中的 $parts^+$ 可以寫成

$$\begin{aligned} &parts^+ :: \mathsf{List}^+\ a \to \mathsf{List}\ (\mathsf{List}^+\ (\mathsf{List}^+\ a)) \\ &parts^+ = foldrn\ (\lambda x \to concat \cdot map\ (extend\ x))\ wrap3\ , \\ &\quad \mathbf{where}\ extend\ x\ (ys : yss) = [[x] : ys : yss, (x : ys) : yss] \\ &\qquad\quad wrap3\ x = [[[x]]]\ . \end{aligned}$$

我們考慮一個定義在非空串列上的簡單演算法推導練習。其次，這也將是一個使用 *foldrn* 與「引入脈絡」的例子，

下述函數 $ascending :: \mathsf{List}^+\ \mathsf{Int} \to \mathsf{Bool}$ 判斷一個串列是否為遞增：

$$\begin{aligned} &ascending\ [x] &&= \mathsf{True} \\ &ascending\ (x : y : xs) &&= x \leqslant y \wedge ascending\ (y : xs)\ . \end{aligned}$$

給定一個整數串列，如何將它切成一個個區段，使得每個區段都是遞增的？如果我們讓每個元素都自己成一段，似乎是滿足需求，但這沒什麼意思。我們希望讓遞增區段盡量連續，也就是說，我們要區段數目最少的分割法。下述函數 *upHills* 將輸入串列以最精簡的方式切成段：

$$\begin{aligned} &upHills :: \mathsf{List\ Int} \to \mathsf{List\ (List\ Int)} \\ &upHills = shortest \cdot filter\ (all\ ascending) \cdot parts^{+}\ . \end{aligned}$$

其中 $parts^{+}$ 把串列任意切段，$filter\ (all\ ascending)$ 挑出每個區段都是遞增的分割法，而 *shortest* 挑選元素數目最少的串列。我們能由此導出一個比較快的演算法嗎？

我們先將 $filter\ (all\ ascending)$ 融入 $parts^{+}$ 之中。經過一些稍繁瑣但原則上並不困難的計算，我們可得：

$$\begin{aligned} &filter\ (all\ ascending) \cdot parts^{+} = \\ &\quad foldrn\ (\lambda x \to concat \cdot map\ (extendAsc\ x))\ wrap3\ . \end{aligned}$$

這和 $parts^{+}$ 的差別只在 *extend* 變成了 *extendAsc*. 後者的定義為：

$$\begin{aligned} extendAsc\ x\ (ys : yss) = \mathbf{if}\ x \geqslant head\ ys\ \mathbf{then}\ &[[x] : ys : yss, (x : ys) : yss] \\ \mathbf{else}\ &[[x] : ys : yss]\ . \end{aligned}$$

函數 *extendAsc* 比 *extend* 多做了一個檢查，只在 $x \geqslant head\ ys$ 時將 $(x : ys) : yss$ 列為一個可能選項。注意：由於 *ys* 的型別是 $\mathsf{List^{+}\ Int}$, *head* 一定可成功。如果我們使用 List Int, 在這裡就得多做些條件判斷。雖然每個非空串列 $\mathsf{List^{+}}$ 都可用 List 表達, 有些問題使用 $\mathsf{List^{+}}$ 描述時會比較便於證明與推論。

接著我們試圖融合 *shortest* 與 $filter\ (all\ ascending) \cdot parts^{+}$. 基底函數為 $(shortest \cdot wrap3)\ x = shortest\ [[[x]]] = [[x]]$. 至於步驟函數，我們希望找到滿足下述融合條件的 *step*:

$$shortest\ (concat\ (map\ (extendAsc\ x)\ ysss)) = step\ x\ (shortest\ ysss)\ .$$

由左手邊開始，由於 *shortest* 可分配進 *concat*（意即 $shortest \cdot concat = shortest \cdot map\ shortest$），我們可推論：

$$\begin{aligned} &shortest\ (concat\ (map\ (extendAsc\ x)\ ysss)) \\ =\ &\quad \{\ shortest\ 分配進\ concat; map\ 融合\ \} \\ &shortest\ (map\ (shortest \cdot extendAsc\ x)\ ysss)\ . \end{aligned}$$

為了有些進展，我們看看 $shortest \cdot extendAsc\ x$ 能如何化簡。將輸入（非空串列）寫成 $ys : yss$:

$$\begin{aligned} &shortest\ (extendAsc\ x\ (ys : yss)) = \\ =\ &\quad \{\ extend'\ 之定義; 提出\ \mathbf{if}\ \} \end{aligned}$$

$$
\begin{aligned}
& \textbf{if } x \geqslant head\ ys \textbf{ then } shortest\ [[x] : ys : yss, (x : ys) : yss] \\
& \qquad\qquad \textbf{else } shortest\ [[x] : ys : yss] \\
= \quad & \{\ shortest \text{ 挑選較短之串列} \ \} \\
& \textbf{if } x \geqslant head\ ys \textbf{ then } (x : ys) : yss \textbf{ else } [x] : ys : yss \quad .
\end{aligned}
$$

因此，融合條件的左手邊可歸約如下：

$$
\begin{aligned}
& shortest\ (map\ (shortest \cdot extendAsc\ x)\ ysss) \\
= \quad & \{\ \text{前述推導} \ \} \\
& shortest\ (map\ (\lambda (ys : yss) \to \textbf{if } x \geqslant head\ ys \textbf{ then } (x : ys) : yss \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } [x] : ys : yss)\ ysss) \quad .
\end{aligned}
$$

我們希望繼續將 *shortest* 往裡推，但此時似乎卡住了。

$$
\begin{aligned}
& shortest\ (map\ (\lambda (ys : yss) \to \textbf{if } x \geqslant head\ (head\ (head\ ysss))\ \textbf{then } (x : ys) : yss \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } [x] : ys : yss)\ ysss) \quad . \\
= \ & shortest\ (\textbf{if } x \geqslant head\ (head\ (head\ ysss)) \\
& \qquad \textbf{then } map\ (\lambda (ys : yss) \to (x : ys) : yss)\ ysss \\
& \qquad \textbf{else } map\ (\lambda (ys : yss) \to [x] : ys : yss)\ ysss) \\
= \ & \textbf{if } x \geqslant head\ (head\ (head\ ysss)) \\
& \quad \textbf{then } shortest\ (map\ (\lambda (ys : yss) \to (x : ys) : yss)\ ysss) \\
& \quad \textbf{else } shortest\ (map\ (\lambda (ys : yss) \to [x] : ys : yss)\ ysss) \\
= \ & \textbf{if } x \geqslant head\ (head\ (head\ ysss)) \\
& \quad \textbf{then } (\lambda (ys : yss) \to (x : ys) : yss)\ (shortest\ ysss) \\
& \quad \textbf{else } (\lambda (ys : yss) \to [x] : ys : yss)\ (shortest\ ysss) \\
= \ & \textbf{let } (ys : yss) = shortest\ ysss \\
& \ \textbf{in if } x \geqslant head\ ys \\
& \quad \textbf{then } (x : ys) : yss \textbf{ else } [x] : ys : yss \quad .
\end{aligned}
$$

$$
\begin{aligned}
& upHills = foldrn\ step\ (\lambda x \to [[x]]) \quad , \\
& \quad \textbf{where } step\ x\ (ys : yss) = \textbf{if } x \geqslant head\ ys \textbf{ then } (x : ys) : yss \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } [x] : ys : yss \quad .
\end{aligned}
$$

習題 6.18 — 將 $filter\ (all\ ascending) \cdot parts^{+}$ 融合為 $foldrn\ (\lambda x \to concat \cdot map\ (extendAsc\ x))\ wrap3$, 並在過程中推導 *extendAsc* 的定義。您可能用得上習題 6.6提及的性質：如果 $filter\ p\ (f\ x) = \textbf{if } p\ x \textbf{ then } g\ x \textbf{ else } [\,]$, 則 $concat \cdot map\ (filter\ p \cdot f) = concat \cdot map\ g \cdot filter\ p$.

6.6 參考資料

Gibbons et al. [2001]

CHAPTER 7

區段問題

一個串列中連續的任意一截被稱作一個「區段」(segment)。例如，$[1,2,3]$ 的區段包括 $[]$,$[1]$,$[2]$,$[3]$,$[1,2]$,$[2,3]$, 以及 $[1,2,3]$本身（注意：空串列也是一個區段）。也許因為歷史因素，許多有趣的演算法問題有這樣的形式：給定一個串列 $xs :: \mathsf{List\ A}$，一個述語 $p :: \mathsf{List\ A} \to \mathsf{Bool}$, 和一個函數 $f :: \mathsf{List\ A} \to \mathsf{B}$，我們想在 xs 的所有區段中，找出滿足述語 p 並使得 f 值最大的那個。我們把這類問題統稱為「區段問題」。

本節討論一些簡單的區段問題。我們先回顧一下區段的形式定義。第2.6.3節曾提及，下述函數 *segments* 算出一個串列的所有區段：

$$\begin{aligned} &segments :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &segments = concat \cdot map\ inits \cdot tails\ . \end{aligned}$$

其中 $tails :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$ 計算一個串列所有的後段 (suffixes)，例如 $tails\ [1,2,3] = [[1,2,3],[2,3],[3],[]]$；函數 $inits :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a)$ 則計算一個串列的所有前段 (prefixes)，例如 $inits\ [1,2,3] = [[],[1],[1,2],[1,2,3]]$. 對每一個後段，計算所有的前段，便得到所有的區段了。[1] 事實上，如此的定義之下 $segments\ [1,2,3]$ 會將空串列傳回很多次。不過以本章的目的而言，我們對這些重複的 $[]$ 並不在意。

為方便讀者，我們將 *inits* 與 *tails* 的歸納定義重複如下。

$$\begin{aligned} &inits :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &inits\ [] \quad\ = [[]] \\ &inits\ (x : xs) = [] : map\ (x:)\ (inits\ xs)\ , \end{aligned} \qquad \begin{aligned} &tails :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\ &tails\ [] \quad\ = [[]] \\ &tails\ (x : xs) = (x : xs) : tails\ xs\ . \end{aligned}$$

[1] 我們也可以反過來，定義 $segments = concat \cdot map\ tails \cdot inits$. 本節中的所有推導與證明可調整成相對應的版本。

7.1 最大區段和

給定一個串列。請問它的眾多區段中，總和最大的和是多少？這個最大區段和(*maximum segment sum*)問題可說是最經典的區段問題。我們可將其寫成如下的規格：

$$\begin{aligned}&mss :: \mathsf{List\ Int} \to \mathsf{Int}\\ &mss = maximum \cdot map\ sum \cdot segments\ \ ,\end{aligned}$$

這幾乎便是問題的字面翻譯：算出所有的區段，對每個區段算其和，然後挑出最大的一個。[2] 當輸入串列長度為 n, 這個定義本身是一個時間複雜度為 $O(n^3)$ 的演算法 — 該串列的區段有 $O(n^2)$ 個，每個都需分別算總和。我們能導出一個更快的演算法嗎？

前段-後段分解 許多區段問題的推導都以如下方式開頭：將 *segments* 展開成 *inits* 與 *tails*, 並將 *maximum* 往右推，與 *inits* 放在一起：

$$\begin{aligned}&maximum \cdot map\ sum \cdot segments\\ =\ &maximum \cdot map\ sum \cdot concat \cdot map\ inits \cdot tails\\ =\ &\quad \{\, map\ f \cdot concat = concat \cdot map\ (map\ f)\ \text{(習題 2.13)}\,\}\\ &maximum \cdot concat \cdot map\ (map\ sum) \cdot map\ inits \cdot tails\\ =\ &\quad \{\, maximum \cdot concat = maximum \cdot map\ maximum\,\}\\ &maximum \cdot map\ maximum \cdot map\ (map\ sum) \cdot map\ inits \cdot tails\\ =\ &\quad \{\, map\ \text{融合（定理 2.2）}\,\}\\ &maximum \cdot map\ (maximum \cdot map\ sum \cdot inits) \cdot tails\ \ .\end{aligned}$$

細看 $maximum \cdot map\ sum \cdot inits$ 這個子算式，其意思是「給定一個串列，計算其所有前段的和的最大值」。我們為這個子算式取個名字，令 $mps = maximum \cdot map\ sum \cdot inits$，其中 *mps* 為「最大前段和 maximum prefix sum」的縮寫。經由上述演算，我們得知

$$mss = maximum \cdot map\ mps \cdot tails\ \ ,$$

意思是：要找出所有區段的最大和，我們可以對每一個後段，找出其最大前段和。

這是解許多區段問題的常見模式：要解決最佳區段問題，先試著解最佳前段問題。要算出最佳區段，可對每一個後段，算出其最佳前段。[3]

使用掃描引理 接下來我們注意到 $map\ mps \cdot tails$ 這個子算式。回顧掃描引理 (6.11), 重複如下：

$$map\ (foldr\ f\ e) \cdot tails = scanr\ f\ e = foldr\ (\lambda x\ ys \to f\ x\ (head\ ys) : ys)\ [e]\ \ .$$

[2] 函數 $maximum :: \mathsf{List\ Int} \to \mathsf{Int}$ 的定義見例 2.12. 我們得假設整數中有個 $-\infty$ 作為 $maximum\ [\,]$ 的結果。若要避免 $-\infty$，可注意到 *inits*, *tails*, 和 *segments* 都不會傳回空串列 — 它們的型別都可寫成更嚴格的 $\mathsf{List}^+\ (\mathsf{List}\ a)$。因此我們可改用例 2.12中的 $maximum^+$. 本節的推導稍加修改後即可適用。

[3] 反過來當然也可以。如果我們定義 $segments = concat \cdot map\ tails \cdot inits$, 此處的策略就變成「對每個前段，算出其最佳後段」。

如果我們能把 mps 變成一個 $foldr$，$map\ mps \cdot tails$ 可改寫為 $scanr$. 如果該摺的步驟函數只花常數時間，我們就有了一個線性時間的演算法了！

如何把 mps 變成摺呢？由於 $inits$ 是摺，我們可使用摺融合。此處的計算和例 6.9很類似，我們可以把 $map\ sum$ 與 $maximum$ 分兩次融合進 $inits$, 也可以一次把 $maximum \cdot map\ sum$ 融合進 $inits$. 這次我們試試看後者。基底值 $maximum\ (map\ sum\ [[\,]]) = 0$. 我們想要尋找滿足融合條件 $maxmium\ (map\ sum\ ([\,] : map\ (x:)\ xss)) = step\ x\ (maximum\ (map\ sum\ xss))$ 的函數 $step$. 計算如下：

$$
\begin{aligned}
& maximum\ (map\ sum\ ([\,] : map\ (x:)\ xss)) \\
= \quad & \{\ map\ 與\ sum\ 之定義\ \} \\
& maximum\ (0 : map\ sum\ (map\ (x:)\ xss)) \\
= \quad & \{\ map\ 融合\ \} \\
& maximum\ (0 : map\ (sum \cdot (x:))\ xss) \\
= \quad & \{\ sum\ 之定義\ \} \\
& maximum\ (0 : map\ ((x+) \cdot sum)\ xss) \\
= \quad & \{\ maximum\ 之定義\ \} \\
& 0 \uparrow maximum\ (map\ ((x+) \cdot sum)\ xss) \\
= \quad & \{\ maximum \cdot map\ (x+) = (x+) \cdot maximum\ \} \\
& 0 \uparrow (x + maximum\ (map\ sum\ xss))\ .
\end{aligned}
$$

因此我們推導出了：

$$
\begin{aligned}
& maximum \cdot map\ sum \cdot inits \\
= \ & maximum \cdot map\ sum \cdot foldr\ (\lambda x\ xss \to [\,] : map\ (x:)\ xss)\ [[\,]] \\
= \quad & \{\ 摺融合定理，融合條件如上\ \} \\
& foldr\ (\lambda x\ s \to 0 \uparrow (x + s))\ 0\ .
\end{aligned}
$$

注意：步驟函數推導的最後一步中，為了將 $maximum$ 往右推，使用了如下的性質

$$maximum \cdot map\ (x+) = (x+) \cdot maximum\ . \tag{7.1}$$

在 (7.1) 的左手邊，我們將一個串列的每個元素都加上 x, 然後取最大值。性質 (7.1) 告訴我們，其實我們可以先取最大值，然後做一個加法即可。這個步驟允許我們在每一步省下了 $O(n)$ 個加法，是使得整個演算法之所以能加速的關鍵一步。而 (7.1) 的證明只需使用例行的歸納法，但其中的關鍵一步需要如下的分配律：

$$x + (y \uparrow z) = (x + y) \uparrow (x + z)\ . \tag{7.2}$$

加法與 $(\uparrow)$的分配率是使我們能有一個線性時間演算法的關鍵性質。

既然 mps 已經是一個摺，我們可以使用掃描引理：

$$
\begin{aligned}
& maximum \cdot map\ sum \cdot segments \\
= \quad & \{\ 上述計算\ \} \\
& maximum \cdot map\ (foldr\ (\lambda x\ s \to 0 \uparrow (x + s))\ 0) \cdot tails
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\text{掃描引理 6.11}\} \\
&\quad maximum \cdot scanr\ (\lambda x\ s \to 0 \uparrow (x + s))\ 0\ .
\end{aligned}
$$

我們得到

$$mss = maximum \cdot scanr\ (\lambda x\ s \to 0 \uparrow (x + s))\ 0\ .$$

這是一個使用線性時間、線性空間的演算法。

藉由程式推導，我們不僅找到了一個較快的演算法，也找出了使得該演算法之所以成立的根本性質。這使我們能輕易將該演算法推廣：不僅是加法與 $(\uparrow)$，該演算法能用在任何滿足 (7.2) 的一組運算元之上。

習題 7.1 — 證明性質 (7.1)。

常數空間 使用掃描引理導出的 *mss* 能在線性時間內對輸入串列的每個後段算出其 *mps*（即「最大前段和」）並存放在一個串列中。方法是使用一個 *scanr* 將串列由右到左走訪一遍，在每一步將 x 與 *mps xs* 的值（存放在串列中）相加，並和 0 比大小。每個 *mps* 值之中最大的那個，便是我們要的答案。在函數語言圈內，關於最大區段和的討論大多到此為止：我們已經有了一個漂亮的線性時間演算法了。

若要再挑惕，這個演算法的不盡滿意之處是需要線性的空間 — *scanr* 會產生一個中間串列，由 *maximum* 消掉。我們有可能把 *maximum* 與 *scanr* 融合，得到一個不產生中間串列的演算法嗎？根據摺融合定理，我們將需要找到滿足以下融合條件的函數 *step*：

$$maximum\ (0 \uparrow (x + head\ ys) : ys) = step\ x\ (maximum\ ys)\ .$$

這顯然做不到：從 *maximum ys* 是無法取出 *head ys* 的。

這時就用得上組對的技巧了 — 既然需要 *head*，就把它一併歸納地算出來吧！定義：

$$msps = \langle maximum, head \rangle \cdot scanr\ (\lambda x\ s \to 0 \uparrow (x + s))\ 0\ .$$

要把 $\langle maximum, head \rangle$ 融入 *scanr*, 我們依融合條件推導：

$$
\begin{aligned}
&\quad \langle maximum, head \rangle\ (0 \uparrow (x + head\ ys) : ys) \\
&= \quad \{\langle \cdot, \cdot \rangle \text{ 之定義}\} \\
&\quad (maximum\ (0 \uparrow (x + head\ ys) : ys), head\ (0 \uparrow (x + head\ ys) : ys)) \\
&= \quad \{maximum \text{ 與 } head \text{ 之定義}\} \\
&\quad ((0 \uparrow (x + head\ ys)) \uparrow maximum\ ys, 0 \uparrow (x + head\ ys)) \\
&= \quad \{\text{取出 } (maximum\ ys, head\ ys)\} \\
&\quad (\lambda m\ s \to ((0 \uparrow (x + s)) \uparrow m, 0 \uparrow (x + s)))\ (maximum\ ys, head\ ys) \\
&= \quad \{\text{取出重複項 } 0 \uparrow (x + s)\} \\
&\quad (\lambda m\ s \to \textbf{let}\ s' = 0 \uparrow (x + s)\ \textbf{in}\ (s' \uparrow m, s'))\ (maximum\ ys, head\ ys)\ .
\end{aligned}
$$

我們得到

指令式版本的最大區段和

根據第 5.6.2節的討論，當輸入串列 $[x_0 \,..\, x_{N-1}]$ 被逆向存放在陣列 X 之中，函數 *msps* 相當於以下的指令式程式：

```
i, m, s := 0, 0, 0;
do i ≠ N → s := 0 ↑ (X i + s);
           m := m ↑ s;
           i := i + 1
od;
return m
```

這在指令式程式推導的領域中也是一個經典範例。

$$
\begin{aligned}
&mss = fst \cdot msps \ \ , \\
&msps :: \mathsf{List\ Int} \to (\mathsf{Int} \times \mathsf{Int}) \\
&msps = foldr\ (\lambda m\ s \to \mathbf{let}\ s' = 0 \uparrow (x+s)\ \mathbf{in}\ (s' \uparrow m, s'))\ (0,0)\ \ .
\end{aligned}
$$

或著，我們把 *msps* 的 *foldr* 定義展開，也許比較容易理解：

$$
\begin{aligned}
msps\ [\,] &= (0,0) \\
msps\ (x : xs) &= \mathbf{let}\ (m,s) = msps\ xs \\
&\qquad\ \ s' = 0 \uparrow (x+s) \\
&\quad\ \mathbf{in}\ \ (s' \uparrow m, s')\ \ .
\end{aligned}
$$

這是一個使用線性時間、常數空間計算最大區段和的演算法。當 $(m,s) = msps\ xs$, m 是 xs 的最大區段和，s 則是 xs 的最大前段和。每考慮一個新元素 x, 最大前段和被更新為 $0 \uparrow (x+s)$ — 如果把 x 接上後仍是正數，$x+s$ 就是最好的前段和，否則最大前段和是空串列的和 0. 新的最大前段和再與 m 比較，得到新的最大區段和。

7.2 最長高原問題

如果一個區段的每個元素都相等，我們稱之為一個「高原 (plateau)」。本節考慮這個問題：給一個串列，找出其中最長的高原的長度。例如當輸入為 $[2,3,3,2,2,2,1,6,6]$ 時，輸出應為 3 — 即最長的高原 $[2,2,2]$ 的長度。以下是本問題的一種可能的規格寫法：

$$
\begin{aligned}
&lp :: \mathsf{List\ Int} \to \mathsf{Int} \\
&lp = maximum \cdot map\ length \cdot filter\ plateau \cdot segments^{+}\ \ ,
\end{aligned}
$$

其中 $segments^{+}$ 算出所有的區段，*plateau* 判斷一個區段是否為高原。過濾出所有的高原後，我們計算每個高原的長度，然後找出最大值。[4]

函數 $segments^{+}$ 和 *segments* 類似，其定義為：

[4] 函數 *lp* 與 *plateau* 可以有更通用的型別 $\mathsf{Eq}\ a \Rightarrow \mathsf{List}\ a \to \mathsf{Int}$, $\mathsf{Eq}\ a \Rightarrow \mathsf{List}\ a \to \mathsf{Bool}$.

區段問題不傳回區段？

為何我們在最大區段和問題中只傳回最大區段的和、在最長高原問題中只傳回長度，而不傳回該區段本身呢？

這是許多演算法題目常見的簡化：當真正的問題是尋找「使得某值最大的聚合資料結構」時，我們常只要求解題者傳回該值，而不是整個聚合資料結構。如此做的好處之一是讓我們能暫時忽略「如何組出需傳回的資料結構」的細節，更專注在問題本身。這使得問題的規格與推導過程都簡單許多。

如果我們真的需要整個區段，我們總能將前述的簡單版程式擴充為傳回資料結構的程式。讀者不妨試試能否將第 ?? 頁的 *mpsAll* 改寫為型別為 List Int → List (Int × List Int) 的程式 — 對每個後段，計算其最佳前段和，以及該前段。您會發現程式結構並沒有改變，只是多了些繁瑣的細節。

另一個理由是：具有最大和的區段可能不只一個。在函數程式推導之中，如果我們要傳回區段，由於規格也是函數，我們似乎必須在規格中就決定傳回哪一個。若希望規格是「傳回任一個具有最大和的區段」，我們會需要更多機制，例如使用關係 (*relation*) 或使用單子。

$$
\begin{aligned}
&segments^+ :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}^+\ a) \\
&segments^+ = concat \cdot map\ inits^+ \cdot tails^+\ \ ,
\end{aligned}
$$

函數 $inits^+$ 與 $tails^+$ 則與 $inits$ 與 $tails$ 類似，但只傳回非空的前後段，定義如下：

$$
\begin{aligned}
&inits^+ :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}^+\ a) \\
&inits^+\ [\,] \quad\ = [\,] \\
&inits^+\ (x : xs) = [x] : map\ (x:)\ (inits^+\ xs)\ ,
\end{aligned}
\qquad
\begin{aligned}
&tails^+ :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}^+\ a) \\
&tails^+\ [\,] \quad\ = [\,] \\
&tails^+\ (x : xs) = (x : xs) : tails^+\ xs\ \ .
\end{aligned}
$$

以上三個函數的回傳型別都是 List (List$^+$ a) — 每個傳回的前/後/區段都是非空的，但並非每個串列都有非空的前/後/區段。函數 lp 仍可接受空串列作為輸入，但（我們等下將看到）在規格中使用 $segments^+$ 將帶來不少方便。我們也可使用 $segments$ 定最長平原問題的規格，只是會增加一些不利於講解的細節。

函數 $plateau$ 可定義如下：

$$
\begin{aligned}
&plateau :: \mathsf{List}^+\ \mathsf{Int} \to \mathsf{Bool} \\
&plateau\ [x] \quad\ = \mathsf{True} \\
&plateau\ (x : y : xs) = x == y \wedge plateau\ (y : xs)\ \ .
\end{aligned}
$$

由於輸入是非空串列，$plateau$ 的定義中考慮的是「有一個元素」和「有兩個以上元素」的兩種情況。

前段-後段分解　和最大區段和問題一樣，我們先嘗試把這個區段問題分解為「對每個後段，計算最佳前段」：

$$
\begin{aligned}
&\quad maximum \cdot map\ length \cdot filter\ plateau \cdot segments^+ \\
&= maximum \cdot map\ length \cdot filter\ plateau \cdot concat \cdot map\ inits^+ \cdot tails^+ \\
&= \quad \{\ 因\ filter\ p \cdot concat = concat \cdot map\ (filter\ p)\ \} \\
&\quad maximum \cdot map\ length \cdot concat \cdot map\ (filter\ plateau \cdot inits^+) \cdot tails^+ \\
&= \quad \{\ 因\ map\ f \cdot concat = concat \cdot map\ (map\ f),\ map\ 融合\ \}
\end{aligned}
$$

$$
\begin{aligned}
& maximum \cdot concat \cdot map\ (map\ length \cdot filter\ plateau \cdot inits^{+}) \cdot tails^{+} \\
= \quad & \{ \text{因 } maximum \cdot concat = maximum \cdot map\ maximum, map \text{ 融合} \} \\
& maximum \cdot map\ (maximum \cdot map\ length \cdot filter\ plateau \cdot inits^{+}) \cdot tails^{+} \quad .
\end{aligned}
$$

嘗試使用掃描引理 下一步：我們把 $maximum \cdot map\ length \cdot filter\ plateau \cdot inits^{+}$ 簡寫為 lpp，並嘗試將它寫成 (??) 的形式，以便使用掃描引理。演算如下：

$$
\begin{aligned}
& (maximum \cdot map\ length \cdot filter\ plateau \cdot inits^{+}\ \$\ (x : xs)) \\
= & (maximum \cdot map\ length \cdot filter\ plateau\ \$\ [x] : map\ (x:)\ (inits^{+}\ xs)) \\
= \quad & \{ filter \text{ 之定義，} plateau\ [x] = \mathsf{True} \} \\
& maximum\ (1 : map\ length\ (filter\ plateau\ (map\ (x:)\ (inits^{+}\ xs)))) \\
= & 1 \uparrow maximum\ (map\ length\ (filter\ plateau\ (map\ (x:)\ (inits^{+}\ xs)))) \quad .
\end{aligned}
$$

演算至此的問題是：如何把 $map\ (x:)$ 提出來呢？定理 2.4允許我們把 map 搬到 $filter$ 的左邊：

$$filter\ p \cdot map\ f = map\ f \cdot filter\ (p \cdot f) \quad .$$

至於 $plateau \cdot (x:)$ 能否再化簡？觀察 $plateau$ 定義的第二個子句：

$$plateau\ (x : y : xs) = x == y \wedge plateau\ (y : xs) \quad ,$$

寫成函數組合的形式便是：

$$plateau \cdot (x:) = ((x ==) \cdot head) \mathbin{\dot\wedge} plateau \quad ,$$

其中 $(\dot\wedge)$ 為「提升成函數版」的 $(\wedge)$, 定義為

$$
\begin{aligned}
& (\dot\wedge) :: (a \to \mathsf{Bool}) \to (a \to \mathsf{Bool}) \to a \to \mathsf{Bool} \\
& (f \mathbin{\dot\wedge} g)\ x = f\ x \wedge g\ x \quad .
\end{aligned}
$$

函數 $(\wedge) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$ 拿兩個布林值、算出一個新布林值，$(\dot\wedge)$ 則拿兩個函數 $f, g :: a \to \mathsf{Bool}$，組合成另一個型別為 $a \to \mathsf{Bool}$ 的函數，其結果是 f 與 g 之傳回值的合取。函數 $filter$ 與 $(\dot\wedge)$ 有如下的性質：

$$filter\ (p \mathbin{\dot\wedge} q) = filter\ p \cdot filter\ q \quad . \tag{7.3}$$

有了以上眾多性質，我們演算如下：

$$
\begin{aligned}
& filter\ plateau \cdot map\ (x:) \\
= \quad & \{ \text{定理 2.4} \} \\
& map\ (x:) \cdot filter\ (plateau \cdot (x:)) \\
= \quad & \{ plateau \text{ 之定義} \} \\
& map\ (x:) \cdot filter\ (((x ==) \cdot head) \mathbin{\dot\wedge} plateau) \\
= \quad & \{ \text{因 (7.3): } filter\ (p \mathbin{\dot\wedge} q) = filter\ p \cdot filter\ q \} \\
& map\ (x:) \cdot filter\ plateau \cdot filter\ ((x ==) \cdot head) \quad .
\end{aligned}
$$

現在整個式子成為 $1\uparrow(maximum \cdot map\ (length \cdot (x:)) \cdot filter\ plateau \cdot filter\ ((x==) \cdot head) \cdot inits^+)\ \$\ xs$.

把 $map\ (x:)$ 往左搬之後，式子的右邊出現了 $filter\ ((x==) \cdot head)\ (inits^+\ xs)$ — 產生所有 xs 的非空前段，取出第一個元素為 x 的。但讀者們可能立刻發現：$inits^+$ 傳回的每個前段的第一個元素都是一樣的！也就是說我們有以下性質。

$$filter\ ((x==) \cdot head)\ (inits^+\ xs) = \\ \quad \textbf{if}\ x == head\ xs\ \textbf{then}\ inits^+\ xs\ \textbf{else}\ [\,]\ . \tag{7.4}$$

考慮非空前段的好處在這兒可看出：$head$ 對非空串列才有值。

回到 lpp，我們可繼續推導如下：

$$\begin{aligned}
&lpp\ (x:xs) \\
=\ &\{\ \text{前述演算, } map \text{ 融合}\ \} \\
&1\uparrow(maximum \cdot map\ (length \cdot (x:)) \cdot filter\ plateau \cdot \\
&\qquad filter\ ((x==) \cdot head) \cdot inits^+\ \$\ (x:xs)) \\
=\ &\{\ \text{因 (7.4)}\ \} \\
&1\uparrow(maximum \cdot map\ (length \cdot (x:)) \cdot filter\ plateau\ \$ \\
&\qquad \textbf{if}\ x == head\ xs\ \textbf{then}\ inits^+\ xs\ \textbf{else}\ [\,]) \\
=\ &\{\ \text{(2.3): 函數分配進 } \textbf{if}\ \} \\
&\textbf{if}\ x == head\ xs \\
&\quad \textbf{then}\ 1\uparrow(maximum \cdot map\ (length \cdot (x:)) \cdot filter\ plateau \cdot inits^+\ \$\ xs) \\
&\quad \textbf{else}\ 1\uparrow(maximum \cdot map\ (length \cdot (x:)) \cdot filter\ plateau\ \$\ [\,]) \\
=\ &\{\ length \cdot (x:) = (1+) \cdot length, \text{ 及其他化簡}\ \} \\
&\textbf{if}\ x == head\ xs \\
&\quad \textbf{then}\ 1 + (maximum \cdot map\ length \cdot filter\ plateau \cdot inits^+\ \$\ xs) \\
&\quad \textbf{else}\ 1 \\
=\ &\{\ lpp \text{ 之定義}\ \} \\
&\textbf{if}\ x == head\ xs\ \textbf{then}\ 1 + lpp\ xs\ \textbf{else}\ 1\ \ .
\end{aligned}$$

由此我們得到

$$\begin{aligned}
lpp\ [x]\ \ \ &= 1 \\
lpp\ (x:xs) &= \textbf{if}\ x == head\ xs\ \textbf{then}\ 1 + lpp\ xs\ \textbf{else}\ 1\ \ .
\end{aligned}$$

然而，我們雖為 lpp 推導出了一個歸納定義，該定義並不符合 (??)！後者要求 lpp 的右手邊必須是 $x \oplus lpp\ xs$ 的形式 — 在 $lpp\ xs$ 之外不能有其他的 xs, 而上述的 lpp 右手邊多了一個 $head\ xs$.

這時組對的技巧又派上用場了。定義：

$$lpph\ xs = (lpp\ xs, head\ xs)\ \ ,$$

我們可推導出

$$\begin{aligned}
lpph\ [x]\ \ \ &= (1,x) \\
lpph\ (x:xs) &= (\textbf{if}\ x == y\ \textbf{then}\ 1+n\ \textbf{else}\ 1, x)\ \ , \\
&\quad \textbf{where}\ (n,y) = lpph\ xs\ \ .
\end{aligned}$$

該函數符合 (??) 的形式，其中 $e = (1,x), x \oplus (y,n) = (\textbf{if}\ x == y\ \textbf{then}\ 1+n\ \textbf{else}\ 1, x)$.

總結 綜合目前為止的推導，函數 *lpp* 的推導大架構如下：

$$
\begin{aligned}
& lpp \\
= & \quad \{\text{前段-後段分解}\} \\
& maximum \cdot map\ (maximum \cdot map\ length \cdot filter\ plateau \cdot inits^{+}) \cdot tails^{+} \\
= & \quad \{\text{前述演算：尋找歸納定義}\} \\
& maximum \cdot map\ lpp \cdot tails^{+} \\
= & \quad \{\text{因 } lpp = fst \cdot lpph\} \\
& maximum \cdot map\ (fst \cdot lpph) \cdot tails^{+} \\
= & \quad \{\text{掃描引理}\} \\
& maximum \cdot map\ fst \cdot lpphAll\ \ ,
\end{aligned}
$$

其中 *lpphAll* 的定義如下：

$$
\begin{aligned}
lpphAll\ [x] \quad &= [(1,x)] \\
lpphAll\ (x:xs) &= (\textbf{if}\ x == y\ \textbf{then}\ 1+n\ \textbf{else}\ 1, x) : (n,y) : ys\ \ , \\
&\quad \textbf{where}\ ((n,y):ys) = lpphAll\ xs\ \ .
\end{aligned}
$$

這是一個使用線性時間、線性空間的演算法。

7.3 參考資料

最大區段和 對試圖推銷程式推導的人來說，最大區段和問題幾乎有模範問題該有的一切特質：目標不難理解，但又不容易一眼看出怎麼解得快；大部分的推導過程都能用很形式化、依符號推導的方式進行；推出的程式有顯著的效率提升。程式僅需短短幾行，乍看之下卻不容易理解為何會正確。因此最大區段和問題是程式推導圈內常常提及的經典例子。

根據 Bentley 的 *Programming Pearls*[Bentley, 1986] 一書，最大區段和問題最初由 Brown 大學的 Ulf Grenander 所提出。他當時正設計一個圖形配對的函式。其中，具有最大區段和的二維子陣列是圖形中最有可能含有指定樣式的區域。二維問題比較難解，因此他先考慮一維的情況：

> *1977* 年，*[Grenander]* 把該問題講給 *UNILOGIC* 公司的 *Michael Shamos* 聽，後者一夜之間就設計出了**演算法 3**. *Shamos* 不久後告訴我這個問題時，我們都認為那大概就是最好的解法了；... 又幾天後，*Shamos* 在 *Carnegie-Mellon* 大學的專題討論會中講了這個問題和它的來龍去脈。當時在場的統計學家 *Jay Kadane* 一聽，幾分鐘內就設計出了**演算法 4**.
>
> — *Jon Bentley,* Programming Pearls *(第一版), 第 76 頁.*

Kadane 的**演算法 4** 就是現在廣為人所知的（指令語言版）線性時間解（見第 167頁）。

Shamos 的**演算法 3** 則是一個採取分而治之 (*divide and conquer*) 法的演算法：將陣列分成長度略等的兩半 $xs +\!\!+ ys$, 分別計算 *xs* 與 *ys* 的最大區段和。但除此之外，還得考慮跨越 *xs* 與 *ys* 的區段。因此該演算法在遞迴的每一層又多用了

Shamos 的最大區段和演算法

以下是我根據個人理解將 Shamos 的**演算法 3** 寫成的 Python 程式。

```
def mss (l,u):
  if l>u:
    return 0              # 空陣列的情況
  else if l == u:
    return (0↑a [l])      # 單一元素陣列的情況
  else:
    m = (l+u) / 2
    # 計算 a[l..m] 的最大後段和
    sum,maxToLeft = 0,0
    for i in range (m,l-1,-1):
      sum = sum+a [i]
      maxToLeft = maxToLeft↑sum
    # 計算 a[m+1..u] 的最大前段和
    sum,maxToRight = 0,0
    for i in range (m+1,u+1):
      sum = sum+a [i]
      maxToLeft = maxToRight↑sum
    maxCrossing = maxToLeft+maxToRight
    # 遞迴計算 a[l..m] 與 a[m+1..u] 的最大區段和
    maxInL = mss (l,m)
    maxInR = mss (m+1,u)
    return (maxInL↑maxCrossing↑maxInR)
```

兩個迴圈，分別計算 xs 的最大後段和與 ys 的最大前段和，兩者之和就是橫跨 xs 與 ys 的最大區段和。（見第 172頁 。）該演算法使用 $O(n \log n)$ 的時間。

事後回顧，Shamos 其實不需在每層都把最大前段與後段和重頭算起。我們可用類似組對的想法，對每個子陣列都由下至上地算出以下四個值：最大前段和、最大區段和、最大後段和，以及總和。如此一來我們可得到一個線性時間演算法。這可看作給我們的一個暗示：「分而治之」在此也許是不必要的，我們不需要把陣列從中切半。事實上，把陣列分成頭與尾反而讓事情簡化不少。Kadane 立刻想出了**演算法 4**，不知是否用了同樣的思路？

我最初是在 Gibbons [1997] 中見到此問題，當時便覺得印象深刻。Gibbons 的這篇文章是很好的函數程式推導簡介。Bird et al. [1996] 改用關係 — 函數的一種擴充 — 解最大區段和問題，並推廣到其他資料結構。Mu [2008] 則以最大區段和問題為開端，討論一些有趣的變形：例如有長度限制的最大區段和，和最大平均區段。

區段問題 Zantema [1992]

CHAPTER 8

單子與副作用

從純函數語言的觀點而言，程式是函數，而函數的「本分」是把一個輸入值對應到一個輸出值，除此之外發生的都是副作用 (side effects)。一個程式執行時若可能改變某個可變變數 (*mutable variable*) 的值，是一種副作用 — 這可能是大家最耳熟能詳的例子。[1] 此外，讀寫檔案、抓滑鼠的位置、往螢幕寫東西、和系統問現在的日期時間、取亂數等等，也是副作用。程式若拋出例外 (exception)、不傳回值了，是一種副作用。一個程式若有不止一個可能的傳回值（程式員無法預測是哪一個），也是一種副作用。在更嚴格的定義中，一個程式如果進入無窮迴圈而不終止（因此不傳回值），也可視為副作用。

許多程式設計典範都大量依靠副作用完成各種工作：更新變數、輸出入、用例外處理執行時間遇到的錯誤...。在大家的一般印象中，純函數語言反其道而行，其特徵就是「不能有副作用」。但這其實是個過度簡化的說法。實情正好相反：純函數語言不僅允許副作用，甚至允許多種副作用並存，並且可把一個程式所被允許的副作用標示在其型別中。只是，副作用也必須被納入嚴謹的數學架構中。在純函數語言中，我們需好好地談每個副作用到底「是什麼」(即其語意為何)，滿足哪些性質，而這些性質便可用來推論含副作用的程式的正確性。

例如，「拋出例外 (exception)」算是一種副作用。如果我們定義如下的型別：

$$\textbf{data}\ \mathsf{Except}\ b = \mathsf{Exception}\ \mathsf{String} \mid \mathsf{Return}\ b\ \ ,$$

那麼一個正常終止時傳回型別為 b 的值、但也有可能拋一個例外（並附帶一個字串錯誤訊息）的程式可視為一個值域為 $\mathsf{Except}\ b$ 的全函數 — 正常終止時傳回

[1] 在數學式或函數語言程式中，例如 $x+2\times y+1$, 其中的 x 和 y 被稱為 "variables"，譯為「變數」。這名稱的由來是：相對於 1, 2 等等常數，「變數」的值被其所屬環境而決定。但在同一個環境與範圍 (scope) 中，一個變數仍代表同一個值。命令式語言中的「變數」則是存放值的容器，我們可使用賦值 (assignment) 指令改變變數中的值。為了區分，我們把後者這種能被賦值而改變內容的變數稱作「可變變數 (mutable variables)」。

Return x, 「拋出例外」則可用傳回 Exception msg 來模擬。有多個可能傳回值（型別均為 b）的程式可視為值域為 Set b 的全函數 — 所有可能的回傳值都被收集在集合之中。假設有一個型別為 s 的可變變數，那麼一個執行完畢後會傳回 b、但可能改變該可變變數的程式可以視為一個型別為 $s \to (b,s)$ 的函數 — 輸入的 s 為該可變變數的初始值，輸出的序對中，b 是程式的結果，s 是可變變數的新值。例如，當 s 為 Int, 一個傳回該可變變數的現有值、並同時將變數加一的「指令」可表示成一個函數 $\lambda s \to (s, 1+s)$.

如此一來，我們可在純函數語言中實作出許多種副作用。這可能和讀者心目中的「實作」相當不同 — 此處的「將可變變數加一」並沒有真正去修改原記憶體位置中的值，而是產生了一個新值；「拋例外」並未採用高效率的實作（例如到系統堆疊中把回傳位置一個個 pop 出來），而和一般函數一樣，只是傳回一個資料結構。但我們可把前述這些較有效率的做法當作交給編譯器做的最佳化 — 當編譯器看到傳回 Except b 的函數，或著看到 $s \to (b,s)$ 這樣的函數，可自己想辦法去找有效率的編譯方式。[2] 身為程式員，我們在乎的是將「會拋例外的程式其實是什麼？會修改可變變數的程式其實是什麼？」給描述出來。

另一個可能令讀者困擾的是：這樣寫程式實在不方便。一個使用 Except 的程式可能得有許多 **case** 將子程式的結果配對拆開。一個程式若由許多型別為 $s \to (b,s)$ 的組成，得手動將序對配對，將輸入與輸出的 s 傳來傳去。這些做法都太低階，我們希望能在抽象一點的層次上工作，方便我們寫程式、讓我們能做抽象些的思考。這個抽象概念必須能適用於種種副作用，抓出這些副作用的共同模式。

在 Haskell 中，描述副作用最常用的抽象概念是單子 (*monad*)。大家常認為單子是學習函數語言碰上的第一個難關。希望讀者讀完本章後，會發現它其實沒那麼難。一般談單子的方式是由下往上的：由例外、可變變數等等特例的實作開始，歸納出這些特例的共同模式，使讀者可發現引出單子這個概念的動機。從第 8.1 節開始，我們也將由這種觀點出發解說單子。但也有另一種由上而下的看法：先討論單子與每個副作用的運算子該滿足什麼性質，才談他們可怎麼實作。我們認為這種看法一樣重要。

例：簡單算式求值 以下的資料結構 Expr 表達一個只有整數、負號、和加法的數學式：

$$\textbf{data}\ \mathsf{Expr} = \mathsf{Num}\ \mathsf{Int} \mid \mathsf{Neg}\ \mathsf{Expr} \mid \mathsf{Add}\ \mathsf{Expr}\ \mathsf{Expr}\ .$$

例如 $-3+4$ 可表達為 Add (Neg (Num 3)) (Num 4). 我們可輕易寫一個歸納定義，將一個 Expr 計算成一個整數：

$$\begin{aligned} &eval :: \mathsf{Expr} \to \mathsf{Int} \\ &eval\ (\mathsf{Num}\ n) \quad = nw \\ &eval\ (\mathsf{Neg}\ e) \quad = -(eval\ e) \\ &eval\ (\mathsf{Add}\ e_0\ e_1) = eval\ e_0 + eval\ e_1\ . \end{aligned}$$

看來這是個再單純也不過的純函數。但只要我們稍加擴充這個數學式型別，情況就變複雜了。如果數學式有除法，分母為零時 $eval$ 就得拋出例外；我們

[2] 實務上這相當不容易...TODO

可以為數學式加入變數，那麼 *eval* 碰到變數時得查閱該變數的值；我們也許想要用一個可變變數幫我們記住 *eval* 做了多少次加法。程式只要稍微變複雜，我們就會開始需要許多種副作用。

我們將用這個例子介紹單子。在接下來的幾個章節中，我們將以不同的方式擴充 Expr 型別以及 *eval* 函。每個擴充需要不同的副作用。我們將由如何模擬出副作用為動機出發，介紹單子的概念。引入單子這種抽象化後，我們會發現我們只需改用不同的單子，就可以在不太改變 *eval* 的主要程式的情況下，為 *eval* 加入新的副作用。因此單子可以視為模組化地加入副作用的方法

8.1 例外處理

假設我們添了一個整數除法運算元：

$$\textbf{data}\ \mathsf{Expr} = \mathsf{Num}\ \mathsf{Int} \mid \mathsf{Neg}\ \mathsf{Expr} \mid \mathsf{Add}\ \mathsf{Expr}\ \mathsf{Expr} \mid \mathsf{Div}\ \mathsf{Expr}\ \mathsf{Expr}\ \ ,$$

我們便碰到了一個小難題：要怎麼處理分母是零的狀況呢？

Haskell 的 *div* 函數可做整數除法。碰到分母為零時，*div* 和大部分語言的除法函數一樣讓整個程式當掉。我們不希望程式當掉，因此得在呼叫 *div* 前先檢查一下分母是否為零。但當分母真為零，*eval* 應該傳回什麼呢？

$$\begin{aligned} \mathit{eval}\ (\mathsf{Div}\ e_0\ e_1) = \textbf{let}\ v_1 &= \mathit{eval}\ e_1 \\ \textbf{in if}\ v_1 \neq 0\ \textbf{then}\ \mathit{eval}\ e_0\ \text{`}\mathit{div}\text{`}\ v_1\ \textbf{else}\ ??? \end{aligned}$$

用 Maybe 表達部份函數 一種看法是，有了除法後，*eval* 對於某些輸入沒有對應的值，因此 *eval* 成了一個部份 (partial) 函數。

操作上，我們希望 Haskell 提供拋出例外的功能，*eval* 無值可傳的時候便拋出例外。在程式的另外某處可能接住這些例外，另行處理。本章開頭提及，一個傳回 Int、但可能拋出例外的函數可用傳回 Except Int 的全函數來模擬。Except 再給一次如下：

$$\textbf{data}\ \mathsf{Except}\ b = \mathsf{Expt}\ \mathsf{String} \mid \mathsf{Return}\ b\ \ .$$

我們把 *eval* 的型別改為 Expr → Except Int, 正常運算結束便傳回 Return, 碰到分母為零的情形則傳回 Expt.

這個版本的 *eval* 該怎麼寫呢？碰到 Num n 自然是傳回 Return n:

$$\mathit{eval}\ (\mathsf{Num}\ n) = \mathsf{Return}\ n\ \ .$$

處理 Neg e 的條款則比以前稍微複雜了。我們得檢查 $\mathit{eval}\ e$ 的結果，若是 Return v，應傳回 $-v$，但也需包上一個 Return. 如果遇到 Expt *msg*, 表示 e 的運算失敗了，我們只好把 Expt *msg* 再往上傳。

$$\begin{aligned} \mathit{eval}\ (\mathsf{Neg}\ e) = \textbf{case}\ \mathit{eval}\ e\ \textbf{of}& \\ \mathsf{Return}\ v &\to \mathsf{Return}\ (-v) \\ \mathsf{Expt}\ \mathit{msg} &\to \mathsf{Expt}\ \mathit{msg}\ \ . \end{aligned}$$

Add 的情況下，程式碼看來更冗長了些：

```
eval (Add e0 e1) = case eval e0 of
                     Return v0 → case eval e1 of
                                   Return v1 → Return (v0 + v1)
                                   Expt msg  → Expt msg
                     Expt msg  → Expt msg  .
```

我們得分別檢查 $eval\ e_0$ 與 $eval\ e_1$ 的結果是 Return 或是 Expt，因此用兩層 **case** 分出了四個狀況 – 雖然它們是重複性很高的程式碼。最後是 Div $e_0\ e_1$ 的情況，我們先計算 e_1, 如果是零的話傳回 Expt, 含錯誤訊息 `"division by zero"`, 否則做除法運算。在這過程中我們也需要許多 **case** ：

```
eval (Div e0 e1) = case eval e1 of
                     Return 0  → Expt "division by zero"
                     Return v1 → case eval e0 of
                                   Return v0 → Return (v0 `div` v1)
                                   Expt msg  → Expt msg
                     Expt msg  → Expt msg  .
```

很自然地，我們會希望把這些例行公事抽象掉。

Except 作為單子　再看一次處理 Neg 的條款：

```
eval (Neg e) = case eval e of
                 Return v  → Return (−v)
                 Expt msg → Expt msg  .
```

我們真正想表達的僅是「把 $eval\ e$ 的結果送給 $\lambda v \to$ Return $(-v)$ – 如果有結果的話」。這只是多加了一點手續的函數應用 (application)。回想 Haskell 有個函數應用運算元 $(\$) :: (a \to b) \to a \to b$. 給定 $f :: a \to b$ 和 $x :: a$, 那麼 $f\ \$\ x$ 就是把 x 餵給 f （也就是 $f\ x$）。也許我們可以自己定義一個函數應用運算元 $(=\!\!<\!\!<)$. 概念上，$f =\!\!<\!\!<\ x$ 仍是把 x 餵給 f，但 x 的型別是 Except a, f 的型別是 $a \to$ Except b, 而「檢查 x 是 Return 或 Expt」的動作就可藏在 $(=\!\!<\!\!<)$ 之中（讀者可把 $(=\!\!<\!\!<)$ 的型別和 $(\$)$ 做比較）：

```
(=<<) :: (a → Except b) → Except a → Except b
f =<< Return x  = f x
f =<< Expt msg = Expt msg  .
```

如此一來，$eval$ (Neg e) 的條款可簡潔地寫成：

```
eval (Neg e) = (λv → Return (−v)) =<< eval e  ,
```

如果定義 $negate\ x = -x$, 上式的右手邊可以更簡潔地寫成 Return $\cdot\ negate =\!\!<\!\!<$ $eval\ e$.

但，目前較常見的習慣是把這個運算子的左右顛倒：參數寫前面，函數寫後面。也就是定義:

$$
\begin{aligned}
&(\gg\!=) : \mathsf{Except}\ a \to (a \to \mathsf{Except}\ b) \to \mathsf{Except}\ b \\
&\mathsf{Just}\ x \quad \gg\!= f = f\ x \\
&\mathsf{Nothing} \gg\!= f = \mathsf{Nothing}\ .
\end{aligned}
$$

運算子 $(\gg\!=)$ 唸作 "bind", 是我們將會細談的重要元素。有了它，函數 *eval* 的前兩個條款可以改寫如下：

$$
\begin{aligned}
&\mathit{eval}\ (\mathsf{Num}\ n) = \mathit{return}\ n \\
&\mathit{eval}\ (\mathsf{Neg}\ e) \ = \mathit{eval}\ e \gg\!= \lambda v \to \mathit{return}\ (-v)\ .
\end{aligned}
$$

我們把 Return 寫成函數：$\mathit{return} = \mathsf{Return}$。等下會解釋為什麼。注意：依照 Haskell 的運算元優先順序，$m \gg\!= \lambda v \to e$ 會被理解成 $m \gg\!= (\lambda v \to e)$. Add 和 Div 的情況也可以用 $(\gg\!=)$ 改寫:

$$
\begin{aligned}
\mathit{eval}\ (\mathsf{Add}\ e_0\ e_1) = \ & \mathit{eval}\ e_0 \gg\!= \lambda v_0 \to \\
& \mathit{eval}\ e_1 \gg\!= \lambda v_1 \to \\
& \mathit{return}\ (v_0 + v_1) \\
\mathit{eval}\ (\mathsf{Div}\ e_0\ e_1) = \ & \mathit{eval}\ e_1 \gg\!= \lambda v_1 \to \\
& \textbf{if}\ v_1 == 0\ \textbf{then}\ \mathit{throw}\ \texttt{"division by zero"} \\
& \textbf{else}\ \mathit{eval}\ e_0 \gg\!= \lambda v_0 \to \\
& \qquad \mathit{return}\ (v_0\ \text{`}\mathit{div}\text{`}\ v_1)\ .
\end{aligned}
$$

此處我們把 Expt 寫成 *throw*.

如此一來，我們的程式變得簡短了一些。例如，$\mathit{eval}\ (\mathsf{Add}\ e_0\ e_1)$ 直覺上彷彿在說「把 $\mathit{eval}\ e_0$ 的結果叫做 v_0, $\mathit{eval}\ e_1$ 的結果叫做 v_1, 然後傳回 $v_0 + v_1$」，至於判斷 $\mathit{eval}\ e_0$ 與 $\mathit{eval}\ v_1$ 到底是 Return 還是 Expt 的動作則藏在 $(\gg\!=)$ 之中了。但引入 $(\gg\!=)$ 這個抽象的用意並不只是把程式變短，而是可推廣到其他的副作用之上。這將是接下來幾小節的主題。

提醒一下：有些讀者可能認為 *eval* 的後兩個條款看來像命令式語言：$\mathit{eval}\ e_1 \gg\!= \lambda v_1 \to ..$ 看來就像是 $v_1 := \mathit{eval}\ e_1$ — 把 $\mathit{eval}\ e_1$ 的值寫到 v_1 中。但其實 $(\gg\!=)$ 並沒有賦值（改變變數現有的值）之意，要做比較的話，它更像是增強版的 **let**. 算式 $p \gg\!= \lambda x \to e$ 就像是 $\textbf{let}\ x = p\ \textbf{in}\ e$. 兩者都是「把 p 的值算出，給予一個名字 x, 然後計算 e」。差別則是：

- **let** 處理純的值：在 $\textbf{let}\ x = p\ \textbf{in}\ e$ 之中，如果 p 的型別是 a, x 的型別也是 a; 算式 e 的型別若是 b, 整個式子的型別也是 b; e 的型別相同。
- $(\gg\!=)$ 處理含有 Except 的值：在 $p \gg\!= \lambda x \to e$ 之中，
  - p 的型別得有 Except. 如果 p 的型別是 Except a, 則 x 的型別是 a;
  - e 的型別得有 Except. 如果 e 的型別是 Except b, 整個式子的型別也是 Except b.
  - 因此 $\lambda x \to e$ 是一個型別為 $a \to \mathsf{Except}\ b$ 的函數。

捕捉例外　最後，我們可以定義一個捕捉例外的運算子。算式 $\mathit{catch}\ p\ \mathit{hdl}$ 嘗試配對 p 的值，如果是 Return x, 就傳回結果 x；如果是 Expt *msg*, 就把錯誤訊息

msg 交給例外處理函數 *hdl*。當 *p* 的型別是 Except *a*, *catch p hdl* 的型別也是 *a*, 而 *hdl* 的型別必須是 String → *a*:

$$
\begin{aligned}
&catch :: \text{Except } a \to (\text{String} \to a) \to a \\
&catch\ (\text{Return } x)\ \ hdl = x \\
&catch\ (\text{Expt } msg)\ hdl = hdl\ msg\ \ .
\end{aligned}
$$

例如，下式嘗試計算 *e* 並把結果轉成字串。如果 *e* 能正常計算，字串會是 `"Result:"` 開頭，否則是 `"Error:"` 開頭：

$$
\begin{aligned}
&catch\ (eval\ e)\ (\lambda v \to \texttt{"Result: "} +\!\!+ show\ v) \\
&\qquad\qquad (\lambda msg \to \texttt{"Error: "} +\!\!+ msg)\ \ .
\end{aligned}
$$

下式則無論如何都傳回 Int, 如果 *e* 之中出現了分母為零的除法，我們便傳回 65536:

$$
catch\ (eval\ e)\ id\ (const\ 65536)\ \ .
$$

8.2 單子與單子律

第 0 章提及，「抽象化」意指抽取出一個問題中最關鍵的元素。一個成功的抽象化能抓到同類問題的共通本質，因此可以推廣到其他相關的情境、問題，用同一套方法表達這些情境、解決這些問題。我們在第 8.1 節發現的便是一個關於計算與副作用的成功抽象化。

我們可以把第 8.1 節中的 Except *a* 理解成一個 *尚待完成的計算*，或是一個 *尚待執行的程式*。用 **case** 去配對它便是把這個計算算出來、把程式執行出來。如果順利執行，計算結果的型別為 *a*, 但也可能產生副作用。而由其型別建構子 Except 可看出此處可能的副作用是「可能拋出例外」。運算子 (>>=) 的功用是將小計算/程式串接起來，組成較大的計算/程式。

推廣出去，Except 其實可以是任意一個型別建構元 *m*: 一個型別為 *m a* 的值表示一個尚待完成的計算，或是一個尚待執行的程式。如果順利執行，計算結果的型別為 *a*, 但執行過程中可能產生副作用。至於究竟是什麼副作用，則看 *m* 是什麼而定。此外，每個 *m* 附帶兩個運算子：

- $return :: a \to m\ a$;
- $(>\!\!>\!\!=) :: m\ a \to (a \to m\ b) \to m\ b$.

它們三者合稱為一個*單子* (*monad*).

給任一個值 $x :: a$, 運算子 *return x* 代表一個純粹傳回 *x*, 沒有副作用的計算。以 *m* = Except 為例，自然的選擇是令 *return* = Return. 我們之後會看到其他例子。

運算子 (>>=) 將小計算串接在一起。在 $p >\!\!>\!\!= f$ 之中，$p :: m\ a$ 是一個回傳型別將為 *a*, 但可能有副作用的計算。如果 *p* 被算出來，其結果會被傳給 *f*. 函數 $f :: a \to m\ b$ 是一個拿到 *p* 的結果後*算出一個計算的計算*. 整個式子 $p >\!\!>\!\!= f$ 的型別也是 *m b*. 這是理解單子程式的一個重點：函數本身沒有副作用，但*函數可以算出含有副作用的程式*。如果我們把 *f* 寫成 λ 表示式，在 $p >\!\!>\!\!= \lambda x \to e$ 之

中，p 的結果被取名為 x，而 $e :: m\ b$ 之中可以使用 x. 此處 $(\gg\!=)$ 可理解成由兩個小計算 p 與 e 組出一個大計算的接著劑，兩個小計算用 x 來溝通。

運算子 $return$ 與 $(\gg\!=)$ 是用來組合出含有副作用的程式的元件，因此得滿足一些該有的性質。正式說來，

定義 8.1 (單子). 一個單子 *(monad)*包含一個型別建構子 m, 以及兩個運算子 $return :: a \to m\ a$ 與 $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$. 它們須滿足以下三條**單子律**:

$$\textbf{右單位律 } \textit{(right identity):} \qquad p \gg\!= return = p \ , \tag{8.1}$$

$$\textbf{左單位律 } \textit{(left identity):} \qquad return\ x \gg\!= f = f\ x \ , \tag{8.2}$$

$$\textbf{結合律 } \textit{(associativity):} \qquad (p \gg\!= f) \gg\!= g = p \gg\!= (\lambda x \to f\ x \gg\!= g) \ . \tag{8.3}$$

右單位律若寫成 $p \gg\!= \lambda x \to return\ x$ 也許更好理解。計算 p 的結果 x 被傳到 $(\gg\!=)$ 的右邊，但後者只是單純地把 x 再傳出。這和只寫 p 得是一樣的。

在左單位律的左手邊 $return\ x \gg\!= f$ 之中，函數 f 接收 $return\ x$ 的結果 — 而 $return\ x$ 的結果就是 x! 因此這和 $f\ x$ 得是一樣的。

最後一條是 $(\gg\!=)$ 的結合律：「用 p 與 f 串接出一個程式，再把 g 串接到右邊」與「把 f 與 g 串接在一起，並把 p 串接在它們左邊」得到的必須是一樣的程式。但由於型別的問題，在後者的情況我們需要一個 x 表示 p 的結果，前者則不用。

每定義一個新的單子，我們需選定一個型別 m，定義 $return$ 與 $(\gg\!=)$ 兩個運算子，然後證明它們滿足單子律。需要推論含單子的程式的性質與正確性時，這三條單子律也是我們會用到的性質。

最後一提：定義 8.1 並不是單子的唯一定義方式。其他等價的定義詳見 [todo: where].

習題 8.1 — 證明第 8.1 節中的定義滿足單子律。意即當 $m = \text{Except}$, $return = \text{Return}$, 而 $(\gg\!=)$ 的定義如第 8.1 節所示時，三條單子律都成立。

單子 class 既然 Haskell 有 type class 機制，我們可以把單子定義為一個 class:

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b  .
```

如此明確規定一個屬於 Monad 的型別建構子必須有 $return$ 與 $(\gg\!=)$. Except 型別則是其中的一個特例：

```
instance Monad Except where
  return          = Return
  Expt msg >>= f  = Expt msg
  Return x >>= f  = f x  .
```

之後我們討論其他的 m 時，也將把它們宣告為 Monad 的特例。但這麼做只是為了兩個方便性：1. 讓不同的 m 可以共用 $return$ 與 $(\gg\!=)$ 兩個符號，不用重

新取名；2. 看到使用 *return* 與 $(\gg\!=)$ 的程式時，可以暫時不指定它們到底使用那一個特例。若非為了這些小方便，單子與 type class 不一定得有關聯。

此外，Haskell 中的 type class 宣告也只保證 *return* 與 $(\gg\!=)$ 的型別是正確的。目前 Haskell 並無法保證使用者定義的 *return* 與 $(\gg\!=)$ 滿足前述的單子律。

副作用特定的運算子 不知讀者是否想起：那麼 *throw* 與 *catch* 呢？他們也應該滿足一些性質，例如，$throw\ msg \gg\!= f = throw\ msg$ （拋出例外後，後面的 f 不會執行）和 $catch\ (throw\ msg)\ hdl = hdl\ msg$ （*throw* 拋出的例外由 *hdl* 處理）應該都是合理以及必要的性質。另一方面，它們是屬於「例外」這個副作用的運算子，其他的副作用並不見得會有它們。反之，別的副作用可能會有該副作用的特定運算子。例如，要談可變變數，可能會有個賦值運算子，這是 Except 沒有的。

由此得知，用於表達副作用時，單子的運算子可分為兩大類：1. *return* 與 $(\gg\!=)$ 是各個副作用共通的，每個單子都必須定義 *return* 與 $(\gg\!=)$, 並確保它們滿足單子律。2. 此外，每個副作用還可能擁有特有的運算子，它們可能也有些該滿足的性質。我們將在介紹各個副作用的小節中看到更多例子。

8.3 讀取單子

為了進入下一個例子，我們為算式型別加上變數：

```
data Expr = Num Int | Neg Expr | Add Expr Expr |
            Var Name | Let Var Expr Expr  ,
type Name = String  .
```

Name 是變數名稱，Var v 是出現在算式中的變數，Let 則用來宣告新的區域變數。例如，以下式子表達的是 **let** $x = 3+1$ **in** $(x+4)+x$:

```
Let "x" (Add (Num 3) (Num 1))
  (Add (Add (Var "x") (Num 4)) (Var "x"))  ,
```

變數 x 的值為 $3+1$, 因此整個算式的值是 $((3+1)+4)+(3+1) = 12$.

8.3.1 變數與環境

有了變數後，我們不能只問「Add (Var "x") (Var "y") 的（整數）值是什麼」了，因為我們不知道 x 和 y 的值。有自由變數的算式得放在一個環境下才能談它的值。所謂「環境」是一個變數到值的函數，告訴我們每個變數的值。我們可以用一個串列表示：[3]

```
type Env = [(Name, Int)]  .
```

例如 Add (Var "x") (Num 4) 在環境 $[("x",3),("y",6)]$ 下的值是 7. 我們也假設有一個「查表」函數：

[3] 或著，我們也可以定義 **type** Env = Name → Maybe Int, 而 $lookup\ env\ x = env\ x$.

$$lookup :: \mathsf{Env} \to \mathsf{Maybe\ Int}\ .$$

例如，當 $env = [(\texttt{"x"},3),(\texttt{"y"},6)]$，$lookup\ env\ \texttt{"x"}$ 是 Just 3，$lookup\ env\ \texttt{"z"}$ 則是 Nothing.

提醒一下讀者：Let 是在宣告區域變數，而不是賦值。例如，下式表達 $\mathbf{let}\ x = 3\ \mathbf{in}\ x + (\mathbf{let}\ x = 4\ \mathbf{in}\ x) + (-x)$：

```
Let "x" (Num 3)
  (Add (Add "x" (Let "x" (Num 4) (Var "x")))
    (Neg (Var "x")))
```

它的值是 $3+4+(-3)$，而不是 $3+4+(-4)$ — 內層宣告的 x 僅是暫時遮蓋到外面的 x。

有了這些鋪陳，我們看看新的 $eval$ 函數該怎麼寫。既然算式要在環境之下才有值，$eval$ 得把環境也納為參數之一：

$$eval :: \mathsf{Expr} \to \mathsf{Env} \to \mathsf{Int}$$

函數 $eval$ 拿一個算式和一個環境，計算該算式的值。新的 $eval$ 中，最初的三個條款基本上是一樣的，只是多了一個參數 env 得往下傳：

$$\begin{aligned}
&eval\ (\mathsf{Num}\ n)\ \ \ \ env = n \\
&eval\ (\mathsf{Neg}\ e)\ \ \ \ env = -(eval\ e\ env) \\
&eval\ (\mathsf{Add}\ e_0\ e_1)\ env = eval\ e_0\ env + eval\ e_1\ env\ .
\end{aligned}$$

碰到變數時，我們到環境中查變數的值：

$$eval\ (\mathsf{Var}\ x)\ env = \mathbf{case}\ lookup\ env\ x\ \mathbf{of}\ \mathsf{Just}\ v \to return\ v\ .$$

這裡的 **case** 算式只處理了 Just 的情形。如果 $lookup$ 傳回的是 Nothing，也就是變數 x 並不在環境 env 中，該怎麼辦呢？我們等下再談。最後，碰到 $\mathsf{Let}\ x\ e_0\ e_1$ 時，我們先把 e_0 的值在 env 這個環境之下算出，然後算 e_1:

$$\begin{aligned}
&eval\ (\mathsf{Let}\ x\ e_0\ e_1)\ env = \\
&\quad \mathbf{let}\ v = eval\ e_0\ env \\
&\quad \mathbf{in}\ eval\ e_1\ ((x,v) : env)
\end{aligned}$$

但計算 e_1 時須使用新的環境 $(x,v) : env$，這讓 e_1 可以用到 x。變數 x 在新環境中的值是 v.

算式的語意是函數　第 ?? 節開頭提及，算式沒有變數、沒有除法時，$eval$ 的型別是 $\mathsf{Expr} \to \mathsf{Int}$ — 此時一個算式的「意思」就是一個整數。有了除法後，為了表示可能拋出例外的函數，$eval$ 的型別變成 $\mathsf{Expr} \to \mathsf{Except\ Int}$. 此時 $\mathsf{Except}\ a$ 代表一個尚待完成、可能有副作用（拋出例外）的計算。

加上變數、考慮環境後，$eval$ 的型別變成了 $\mathsf{Expr} \to (\mathsf{Env} \to \mathsf{Int})$：$eval$ 拿一個算式，傳回一個函數；該函數又拿一個環境，才算出一個整數值。也就是說，一個算式的語意是「拿一個環境，傳回一個整數的函數」。的確，既然算

式算成的那個整數必須由環境決定，算式其實不能看作一個數值，而應該是從環境到整數的函數才對。算式 *eval* (Add (Var "x") (Var "y")) 是一個函數，如果給它 [("x",3),("y",2)], 我們得到 5; 如果給 [("x",4),("y",−3)], 我們得到 1.

如果把「給一個環境，傳回型別為 a 的結果的函數」叫做 Reader a, 也就是說定義 **type** Reader a = Env → a，函數 *eval* 的型別成了 Expr → Reader a. 此處，Reader a 也可視為一個尚待完成、可能發生副作用的計算。此處可能的副作用包括「和環境詢問變數的值」。習慣上，我們把這種副作用稱作「讀者 (reader)」或「讀取」。「給環境」的動作，就是執行運算，把值算出來。我們把其中兩個條款改寫成 λ 算式：

$$\begin{aligned} &eval\ (\mathsf{Num}\ n) &&= \lambda env \to n \\ &eval\ (\mathsf{Add}\ e_1\ e_2) &&= \lambda env \to eval\ e_1\ env + eval\ e_2\ env \end{aligned}$$

可以理解成：*eval* (Num n) 傳回一個計算，無論環境為何，該計算都傳回 n; *eval* (Add e_1 e_2) 也傳回一個計算，該計算拿到環境後，在同一個環境之下把 *eval* e_1 和 *eval* e_2 算成值，然後傳回他們的和。

然而，手動把 *env* 傳來傳去是個重複性高、容易出錯、也嫌累贅的動作。在第 8.1 節中，我們把重複性地產生與拆開 Except 型別的動作抽象成 *return* 和 (>>=)，使得 Except 成為一個單子。對於本節的 Reader, 我們也能設計出一組 *return* 和 (>>=), 把重複的動作抽象掉，使得 Reader 成為一個單子嗎？

8.3.2 「讀取」副作用是單子

讀取單子 如第 8.2 節所述，*return* 製作一個沒有副作用的計算，(>>=) 則用於把兩個計算接起來。把 m 代換成 Reader, 它們的型別分別是：

$$\begin{aligned} &return :: a \to \mathsf{Reader}\ a \\ &(\gg\!=) :: \mathsf{Reader}\ a \to (a \to \mathsf{Reader}\ b) \to \mathsf{Reader}\ b\ , \end{aligned}$$

其中 Reader a = Env → a.

函數 *return* 比較單純：*return* x 是一個無論 *env* 是什麼，都傳回 x 的計算：

$$return\ x\ env = x\ .$$

至於 $m \gg\!= f$ 則可定義如下：

$$(p \gg\!= f)\ env = f\ (p\ env)\ env\ .$$

$p \gg\!= f$ 的型別為 Reader b, 意謂它可以收一個環境 *env* 當參數，而等號右手邊必須算出一個型別為 b 的結果。變數 $p ::$ Reader a 是一個尚待完成的計算，p *env* 把它算出來，得到型別為 a 的結果。函數 f 的型別為 $a \to$ Reader b, 因此 $f\ (p\ env)$ 意謂 f 得到 $p\ env :: a$ 的結果，並算出一個新的計算. 這個計算又在得到 *env* 之後才真正被算出來，最後型別為 b.

有了 *return* 與 (>>=), *eval* 的前三個條款可改寫如下：

$$\begin{aligned} &eval :: \mathsf{Expr} \to \mathsf{Reader\ Int} \\ &eval\ (\mathsf{Num}\ n) &&= return\ n \end{aligned}$$

$$
\begin{aligned}
&\mathit{eval}\ (\mathsf{Neg}\ e) &&= \mathit{eval}\ e \gg\!= \lambda v \to \mathit{return}\ (-v)\ .\\
&\mathit{eval}\ (\mathsf{Add}\ e_0\ e_1) &&= \mathit{eval}\ e_0 \gg\!= \lambda v_0 \to\\
&&&\quad \mathit{eval}\ e_1 \gg\!= \lambda v_1 \to\\
&&&\quad \mathit{return}\ (v_0 + v_1)\ ,
\end{aligned}
$$

除了 Except 變成 Reader 之外和第 8.1 節完全相同！由於使用了單子這個適當的抽象，只要把型別從 Except 變成 Reader，主程式在不需大幅更動的情況下就可以沿用。使用單子，我們能模組化地挑選我們想要的副作用。

讀取單子的特定運算　本節新添加的 Var 與 Let 兩個情況是第 8.1 節沒有的。要處理它們，需要讀取副作用特有的運算子。

處理 Var x 時，eval 得到環境中查變數 x 的值。我們姑且先為此專門定義一個函數：

$$
\begin{aligned}
&\mathit{lookupVar} :: \mathsf{Var} \to \mathsf{Reader\ Int}\\
&\mathit{lookupVar}\ x\ \mathit{env} = \mathbf{case}\ \mathit{lookup}\ \mathit{env}\ x\ \mathbf{of}\ \mathsf{Just}\ v \to \mathit{return}\ v\ ,
\end{aligned}
$$

之後再考慮更通用的情況。有了它，eval 遇上 Var x 的條款可寫成：

$$
\mathit{eval}\ (\mathsf{Var}\ x) = \mathit{lookupVar}\ x\ .
$$

計算 Let $x\ e_1\ e_2$ 時，我們需要更動環境，把「x 的值是 e_1 的值」這個資訊加到環境中，並在新環境內執行 $\mathit{eval}\ e_2$, 但執行完之後就回到原有的環境 – 新環境只是局部的。由於這是常見的模式，我們可定義一個更通用的運算子。給定一個函數 $f :: \mathsf{Env} \to \mathsf{Env}$ 用於製作局部的環境，如果目前的環境是 env, 算式 $\mathit{local}\ f\ p$ 在新環境 $f\ \mathit{env}$ 之下執行 p:

$$
\begin{aligned}
&\mathit{local} :: (\mathsf{Env} \to \mathsf{Env}) \to \mathsf{Reader}\ a \to \mathsf{Reader}\ a\\
&\mathit{local}\ f\ p\ \mathit{env} = p\ (f\ \mathit{env})\ .
\end{aligned}
$$

有了 local 的幫忙，eval 遇上 Let 時可寫成：

$$
\begin{aligned}
&\mathit{eval}\ (\mathsf{Let}\ x\ e_0\ e_1) = \mathit{eval}\ e_0 \gg\!= \lambda v \to\\
&\qquad\qquad \mathit{local}\ ((x, v) :)\ (\mathit{eval}\ e_1)\ .
\end{aligned}
$$

我們用 $((x, v) :)$ 幫環境增加一筆資料，在這之下計算 $\mathit{eval}\ e_2$.

習題 8.2 – 使用本節的定義，將 $\mathit{eval}\ (\mathsf{Add}\ e_0\ e_1)\ \mathit{env}$ 與 $\mathit{eval}\ (\mathsf{Let}\ x\ e_0\ e_1)\ \mathit{env}$ 展開，確認它們和第 8.3.1 節的定義一樣。意即：

$$
\begin{aligned}
&\mathit{eval}\ (\mathsf{Add}\ e_0\ e_1)\ \ \mathit{env} = \mathit{eval}\ e_0\ \mathit{env} + \mathit{eval}\ e_1\ \mathit{env}\ ,\\
&\mathit{eval}\ (\mathsf{Let}\ x\ e_0\ e_1)\ \mathit{env} = \mathit{eval}\ e_1\ ((x, \mathit{eval}\ e_0\ \mathit{env}) : \mathit{env})\ .
\end{aligned}
$$

更通用的環境單子 為了說明方便，目前為止我們假設「環境」是某個固定的型別: Env. 我們當然可以把這部份也抽象掉：

```
type Reader e a = e → a .
```

現在 Reader 多吃一個參數 e, 代表環境的型別。對任意的 e, Reader e 都是一個單子，函數 *local* 更通用的型別是 $(e \to e) \to \text{Reader } e\ a \to \text{Reader } e\ a$. 本節之前的每個 Reader 都需代換成 Reader Env. 例如 *eval* 的型別是 Expr → Reader Env Int. 但改變的都只有型別，程式不需變動。

函數 *lookupVar* :: Var → Reader Env Int 只能在 e 為 Env 的情況下運作。更一般來說，讀取單子通常會有一個運算子 *ask*, 把整個環境傳出來供我們使用：

```
ask :: Reader e e
ask env = env .
```

函數 *lookupVar* 則可改用 *ask* 定義為：

```
lookupVar :: Var → Reader Int
lookupVar x = ask >>= λenv →
                case lookup env x of Just v → return v .
```

最後，本節之中讓 Reader e 與前一節的 Except 共用 *return*, (>>=) 等符號。若需在 Haskell 中如此，我們得將 Reader e 宣告為 type class Monad 的一個特例。但由於一些型別檢查的技術問題，Haskell 不允許用 **type** 宣告的別名成為 type class 特例。我們得把 Reader 用 **data** 宣告成一個資料型別：[4]

```
data Reader e a = Reader (e → a) ,
instance Monad (Reader e) where
  return a          = Reader (λe → a)
  Reader r >>= f = Reader (λe → f (r e) e) .
```

8.3.3 推論讀取單子程式的性質

給一個單子程式，如何討論它的性質？比如說，對某一個 e, 如何得知 *eval* (Let "x" (Num 3) (Let "y" (Num 4) e)) 的值是什麼？它和 *eval* (Let "y" (Num 4) (Let "x" (Num 3) e)) 的值是否總是相等？

由於 *eval* 是用 *return*, (>>=), *local* ... 等等運算子定義出來的，而這幾個運算子的定義也都已經有了，我們總是可以把他們的定義都展開，回到最基礎的層次證明任何我們想確認的性質。但一來如此的證明可能非常瑣碎，二來 *return*, (>>=) 等運算子的定義可能還會改變。我們是否能在稍微抽象一點的層次運作，假裝我們不知道這些單子運算子的定義，只討論它們具有什麼性質，並用這些性質來做證明？

[4] 更普遍的做法是用 **newtype** 宣告：**newtype** Reader $e\ a$ = Reader { *runReader* :: $(e \to a)$ }. 但本書不談 **newtype**.

我們應可以合理要求一個「正確」的讀取單子實作應該要滿足下列的性質。首先，假設 e 是一個不含變數 v 的算式：

$$ask \gg\!= \lambda v \to return\ e = return\ e \quad \text{，如果 } v \text{ 不出現在 } e \text{ 之中。} \tag{8.4}$$

等號兩邊都只是傳回 e 的值，而 e 的值不受 v 影響，因此 ask 是可以省略掉的。其次，連續使用 ask 兩次可以縮減為一次就好：

$$ask \gg\!= \lambda v_0 \to ask \gg\!= \lambda v_1 \to f\ v_0\ v_1 \quad = \quad ask \gg\!= \lambda v \to f\ v\ v \quad . \tag{8.5}$$

在等號左手邊，我們把問了環境兩次之後的計算抽象為一個函數呼叫 $f\ v_0\ v_1$. 在右手邊我們則讓 f 的兩個參數都是 v — 詢問環境一次的結果。

下面兩個式子討論當 $local$ 遇上 $return$ 與 $(\gg\!=)$ 時會如何：

$$local\ g\ (return\ e) = return\ e \tag{8.6}$$

$$local\ g\ (p \gg\!= f) = local\ g\ p \gg\!= (local\ g \cdot f) \tag{8.7}$$

在 (8.6) 的左手邊，改變環境之後立刻 $return\ e$, 其實和單純做 $return\ e$ 一樣。性質 (8.7) 則告訴我們 $local\ g$ 可以分配到 $(\gg\!=)$ 的兩側。由於型別之故，$(\gg\!=)$ 的左邊是 $local\ g\ p$, 右邊則得用函數合成 $(\cdot)$.

最後，下列性質將 $local$ 與 ask 關聯在一起：

$$local\ g\ ask = ask \gg\!= (return \cdot g) \quad . \tag{8.8}$$

在 $local\ g$ 的環境之下做 ask, 相當於先做 ask, 然後把得到的環境交給 g 加工。我們可說算式 (8.8) 藉由這兩個運算子的互動定義出了 $local$ 的「意思」。

有了這些性質，我們不需引用 $local$, ask, $(\gg\!=)$ ... 等等的定義，也可論證讀取單子程式的性質了。例如，我們來看看 $\mathbf{let}\ x = 4\ \mathbf{in}\ x + x$ 的值會是什麼：

$$\begin{aligned}
& eval\ (\text{Let "x" (Num 4) (Add (Var "x") (Var "x"))}) \\
= \quad & \{\ eval \text{ 之定義 }\} \\
& eval\ (\text{Num 4}) \gg\!= \lambda v \to local\ (("x", v) :)\ (eval\ (\text{Add (Var "x") (Var "x")})) \\
= \quad & \{\ eval \text{ 之定義 }\} \\
& return\ 4 \gg\!= \lambda v \to local\ (("x", v) :)\ (eval\ (\text{Add (Var "x") (Var "x")})) \\
= \quad & \{\ \text{單子律：左單位律 (8.2)}\ \} \\
& local\ (("x", 4) :)\ (eval\ (\text{Add (Var "x") (Var "x")})) \\
= \quad & \{\ eval \text{ 之定義 }\} \\
& local\ (("x", 4) :)\ (eval\ (\text{Var "x"}) \gg\!= \lambda v_0 \to \\
& \qquad\qquad eval\ (\text{Var "x"}) \gg\!= \lambda v_1 \to return\ (v_0 + v_1)) \\
= \quad & \{\ (8.7)\ \} \\
& local\ (("x", 4) :)\ (eval\ (\text{Var "x"})) \gg\!= \lambda v_0 \to \\
& local\ (("x", 4) :)\ (eval\ (\text{Var "x"})) \gg\!= \lambda v_1 \to \\
& local\ (("x", 4) :)\ (return\ (v_0 + v_1))
\end{aligned}$$

我們將 $local\ (("x", 4) :)\ (eval\ (\text{Var "x"}))$ 抽出來化簡：

$$\begin{aligned}
& local\ (("x", 4) :)\ (eval\ (\text{Var "x"})) \\
= \quad & \{\ eval \text{ 與 } lookupVar \text{ 之定義 }\}
\end{aligned}$$

$$
\begin{aligned}
& \mathit{local}\ (("\texttt{x}",4):)\ (\mathit{ask} \gg\!= \lambda \mathit{env} \to \mathbf{case}\ \mathit{lookup}\ \mathit{env}\ "\texttt{x}"\ \mathbf{of}\ \mathsf{Just}\ v \to \mathit{return}\ v) \\
= \quad & \{\,(8.7)\,\} \\
& \mathit{local}\ (("\texttt{x}",4):)\ \mathit{ask} \gg\!= \lambda \mathit{env} \to \\
& \mathit{local}\ (("\texttt{x}",4):)\ (\mathbf{case}\ \mathit{lookup}\ \mathit{env}\ "\texttt{x}"\ \mathbf{of}\ \mathsf{Just}\ v \to \mathit{return}\ v) \\
= \quad & \{\,(8.8),\ 左單位律\ (8.2)\,\} \\
& \mathit{local}\ (("\texttt{x}",4):)\ (\mathbf{case}\ \mathit{lookup}\ ((\texttt{"x"},4) : \mathit{env})\ "\texttt{x}"\ \mathbf{of}\ \mathsf{Just}\ v \to \mathit{return}\ v) \\
= \quad & \{\,\mathit{lookup}\ ((\texttt{"x"},4) : \mathit{env})\ "\texttt{x}" = \mathsf{Just}\ 4\,\} \\
& \mathit{local}\ (("\texttt{x}",4):)\ (\mathit{return}\ 4) \\
= \quad & \{\,(8.6)\,\} \\
& \mathit{return}\ 4\ \ .
\end{aligned}
$$

由此我們得知 $\mathit{local}\ (("\texttt{x}",4):)\ (\mathit{eval}\ (\mathsf{Var}\ "\texttt{x}"))$ 就是 $\mathit{return}\ 4$. 將它放回原式中：

$$
\begin{aligned}
& \mathit{local}\ (("\texttt{x}",4):)\ (\mathit{eval}\ (\mathsf{Var}\ "\texttt{x}")) \gg\!= \lambda v_0 \to \\
& \mathit{local}\ (("\texttt{x}",4):)\ (\mathit{eval}\ (\mathsf{Var}\ "\texttt{x}")) \gg\!= \lambda v_1 \to \\
& \mathit{local}\ (("\texttt{x}",4):)\ (\mathit{return}\ (v_0 + v_1)) \\
= \quad & \{\,前述計算\,\} \\
& \mathit{return}\ 4 \gg\!= \lambda v_0 \to \mathit{return}\ 4 \gg\!= \lambda v_1 \to \mathit{return}\ (v_0 + v_1) \\
= \quad & \{\,單子律：左單位律\ (8.2)\,\} \\
& \mathit{return}\ (4+4)\ \ .
\end{aligned}
$$

因此，$\mathit{eval}\ (\mathsf{Let}\ "\texttt{x}"\ (\mathsf{Num}\ 4)\ (\mathsf{Add}\ (\mathsf{Var}\ "\texttt{x}")\ (\mathsf{Var}\ "\texttt{x}")))$ 就是 $\mathit{return}\ 8$.

提醒讀者注意一點：$\mathit{eval}\ (\mathsf{Let}\ "\texttt{x}"\ (\mathsf{Num}\ 4)\ (\mathsf{Add}\ (\mathsf{Var}\ "\texttt{x}")\ (\mathsf{Var}\ "\texttt{x}")))$ 和 $\mathit{return}\ 8$ 都不是基礎型別，而是「尚待完成的計算」。論證單子程式時我們常常不是在基礎型別的層次上運作，而是證明一個計算與另一個計算是等價的。這意味著它們傳回同樣的值，也發生同樣的副作用。我們日後會看到更多此類的例子。

習題 8.3 — 證明對所有 $e :: \mathsf{Expr}$,

$$
\begin{aligned}
& \mathit{eval}\ (\mathsf{Let}\ "\texttt{x}"\ (\mathsf{Num}\ 3)\ (\mathsf{Let}\ "\texttt{y}"\ (\mathsf{Num}\ 4)\ e)) = \\
& \quad \mathit{eval}\ (\mathsf{Let}\ "\texttt{y}"\ (\mathsf{Num}\ 4)\ (\mathsf{Let}\ "\texttt{x}"\ (\mathsf{Num}\ 3)\ e))\ \ ,
\end{aligned}
$$

如果 $((\texttt{"x"},3):) \cdot ((\texttt{"y"},4):) = ((\texttt{"y"},4):) \cdot ((\texttt{"x"},3):)$ 成立。

習題 8.4 — 然而，$((\texttt{"x"},3):) \cdot ((\texttt{"y"},4):) = ((\texttt{"y"},4):) \cdot ((\texttt{"x"},3):)$ 並不成立。

我們稍早曾遇到這個問題：如果給這樣的式子 $\mathit{eval}\ (\mathsf{Var}\ "\texttt{x}")\ [(\texttt{"y"},0)]$, 變數 x 並不在環境中，lookup 將傳回 Nothing，這時該怎麼辦？

我們可以再擴充 Reader 的型別，讓 eval 也可以傳回一個 Except 結果：

$$\mathbf{type}\ \mathsf{ReaderExcept}\ e\ a = e \to \mathsf{Except}\ a$$

而 return 和 $(\gg\!=)$ 也得隨之擴充：

$$
\begin{aligned}
& \mathit{return}\ a = \lambda \mathit{env} \to \mathsf{Just}\ a \\
& \mathit{rm} \gg\!= f = \lambda \mathit{env} \to \mathbf{case}\ \mathit{rm}\ \mathit{env}\ \mathbf{of}
\end{aligned}
$$

```
Just v → f v env
Nothing → Nothing
```

這個 ReaderExcept 型別綜合了讀取單子與例外單子的功能，其 *return* 與 $(\gg=)$ 定義也像是兩個單子定義的混合。但這並不是令人相當滿意的做法。ReaderExcept 比起 Reader 又更複雜了一點。日後我們也許會想要有更多功能，例如狀態、輸出入等。ReaderExcept 已經夠抽象難解了，我們並不希望設計、維護越來越龐大的單子。

既然 Maybe 單子讓一個程式加上「例外」的副作用，讀取單子讓一個程式加上可存取環境的功能，我們能否把這兩項功能分別模組化地加入呢？

也就是說，給了兩個單子 M1 和 M2, 能否把他們的功能加在一起，產生另一個新單子呢？

8.4 狀態單子

沿用第 8.1 節之中的型別：

```
data Expr = Num Int | Neg Expr | Add Expr Expr | Div Expr Expr  .
```

這次我們暫且忽略分母為零的可能，而考慮另一個應用：我們想知道計算過程中做了多少次除法。指令式語言中，一個常見的作法是用一個可變變數來計數。在函數語言中我們怎麼模擬這種行為呢？

在指令式語言中，可變變數的值常用來表示整個系統目前的「狀態 (state)」：諸如已處理過的資料個數、處理中的元素號碼、棋盤上每個棋子的位置... 等等。因此我們把「存取可變變數」的能力稱作狀態 (*state*) 副作用。

經過前幾節的熟悉，我們現在應可以更抽象地、由上而下地想像單子了：對於一個我們需要的副作用，先假設其單子存在，考慮它該有哪些運算子、這些運算子該滿足什麼性質、用它們如何寫程式。然後才考慮這個單子的實作。對於狀態這個副作用，我們該怎麼設計其運算子呢？

在命令式語言中，$x := x + 1$ 這行指令把可變變數 x 的值遞增。這一行看似單純的指令其實包含幾項特性：

- 變數是有名字的；
- 當變數名字出現在 $:=$ 的右手邊，表示讀取變數的值；
- 當變數名字出現在 $:=$ 的左手邊，表示將值寫入該變數。

函數語言中討論狀態副作用時偏好用單純些、使用時比較繁瑣、但比較好討論性質的設計：變數沒有名字，並且把「讀取」與「寫入」兩個動作分為單獨的運算子。

具體說來，令 State s a 表示一個結果型別為 a, 但在執行時可讀寫一個型別為 s 的可變變數的計算。我們要求 State s 是一個單子 — 意謂存在著滿足單子律的 $return :: a \to \text{State } s\ a$ 和 $(\gg=) :: \text{State } s\ a \to (a \to \text{State } s\ b) \to \text{State } s\ b$。一個型別為 State s a 的運算中隱藏的可變變數沒有名字，只能用下述兩個運算子存取：

```
get :: State s s  ,
put :: s → State s ()  .
```

運算子 *get* 和第 8.3 節中的 *ask* 類似，讀出該可變變數的值；*put e* 則把 *e* 的值寫到可變變數中，並傳回 ()。例如，當 $s = \mathsf{Int}$, 我們可以定義如下的操作 *inc*，把可變變數加一：

$$\begin{aligned} &inc :: \mathsf{State\ Int}\ () \\ &inc = get \gg\!= \lambda v \to put\ (1 + v)\ . \end{aligned}$$

關於 *put* 有幾件事情可提醒。首先，當我們呼叫 *put e*, 其中的 *e* 已經是一個型別為 *s* 的純數值，不含副作用。例如在 *inc* 之中，我們必須先將變數的值 *get* 出來（這是一個有副作用的動作），才能計算新的值並寫回去。這讓程式寫起來很繁瑣，但也使得推論程式的性質容易許多。其次，關於 *put e* 的型別 $\mathsf{State}\ s\ ()$. 回想：() 是一個只有一個值的型別，該值在 Haskell 中也寫作 ()。給定 $e :: s$ 後，$put\ e :: \mathsf{State}\ s\ ()$ 是一個會存取型別為 *s* 的可變變數，並傳回 () 的計算。在此，「傳回 ()」可理解為傳回一個不帶資訊的值，僅表達「我做完了」。

這個傳回值既然沒有資訊，通常不會被用上。因此當 *put* 不是函數的最後一個動作時，程式常有如下的模樣：

$$\ldots put\ e \gg\!= \backslash\_ \to \ldots$$

由於「執行一段程式，但只需要它的副作用，不需要它的結果」這件事在本章還會常常發生，我們另外訂一個運算元：

$$\begin{aligned} &(\gg) :: m\ a \to m\ b \to m\ b \\ &p \gg q = p \gg\!= \backslash\_ \to q\ . \end{aligned}$$

如此一來 $put\ e \gg\!= \backslash\_ \to q$ 可以簡寫成 $put\ e \gg q$. （注意：$p \gg q$ 並不要求 *p* 的傳回值型別為 ().）

回到 *get* 與 *put*。如果它們的某個實作確實表達了我們前面口述的意圖，該實作應該會滿足以下的規則：

$$\textbf{get-put:} \qquad get \gg\!= put = return\ ()\ , \tag{8.9}$$

$$\textbf{put-get:} \qquad put\ e \gg get = put\ e \gg return\ e\ , \tag{8.10}$$

$$\textbf{put-put:} \qquad put\ e_0 \gg put\ e_1 = put\ e_1\ , \tag{8.11}$$

以及一個和第 8.3 節中的 *ask* 類似的性質：

$$\textbf{get-get:}\ get \gg\!= \lambda v_0 \to get \gg\!= \lambda v_1 \to f\ v_0\ v_1\ =\ get \gg\!= \lambda v \to f\ v\ v\ . \tag{8.12}$$

其中，**get-put** 意謂：將可變變數的值讀出後立刻寫入相當於什麼都不做。規則 **put-get** 意謂：剛做完 *put e* 之後立刻讀取可變變數的值，必定得到 *e*. 規則 **put-put** 意謂：連做兩個 *put* 之後，只有第二個 *put* 寫入的值會被留下。規則 **get-get** 則意謂：連做兩次 *get* 所得到的值必定相同，可以只 *get* 一次就好。

如前所述，用 *get* 與 *put* 可定義出 *inc*, 而我們只需在 *eval* 遇到 Div 的情況中呼叫 *inc*，就可記錄做了多少次除法了：

$$\begin{aligned} eval\ (\mathsf{Div}\ e_0\ e_1) = eval\ e_1 \gg\!= \lambda v_1 \to \\ eval\ e_0 \gg\!= \lambda v_0 \to \end{aligned}$$

$$
\begin{aligned}
&\mathit{inc} \gg \\
&\mathit{return}\ (v_0\ \text{`}\mathit{div}\text{`}\ v_1)\ \ .
\end{aligned}
$$

函數 *eval* 的其他條款可完全不變，只需把所用單子的型別改成 State Int. 由於單子捕捉了含副作用的程式的共通模式，我們只需選用不同的副作用運算子，就可在不更動大部分程式的情況下使用我們需要的副作用。

8.4.1 河內塔問題

本節以河內塔問題為例子，示範狀態單子的推論。

$$
\begin{aligned}
\mathit{hanoi}\ 0 \quad &= \mathit{return}\ () \\
\mathit{hanoi}\ (\mathbf{1}_+\ n) &= \mathit{hanoi}\ n \gg \mathit{inc} \gg \mathit{hanoi}\ n\ \ .
\end{aligned}
$$

$$\mathit{hanoi}\ n = \mathit{put}\ (2 \uparrow n - 1).$$

8.5 參考資料

Wadler [1992]

Bibliography

J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

R. C. Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley & Sons, Ltd., 2003.

R. C. Backhouse. *Algorithmic Problem Solving*. Wiley, 2011.

J. L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, number 36 in NATO ASI Series F, pages 5–42. Springer-Verlag, 1987.

R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998. ISBN 0-13-484346-0.

R. S. Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

R. S. Bird, O. de Moor, and P. F. Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

H. B. Curry. Some philosophical aspects of combinatory logic. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 85–101. North-Holland, 1980.

E. W. Dijkstra. Programming as a discipline of mathematical nature. *American Mathematical Monthly*, 81(6):608–612, May 1974. EWD 361.

E. W. Dijkstra. The notational conventions I adopted, and why. EWD 1300, 2000.

E. W. Dijkstra. The next fifty years. EWD 1243, 2004.

E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

E. W. Dijkstra and A. J. M. van Gasteren. On naming. AvG 67 / EWD 958, 1986.

F. L. G. Frege. Function and concept. translated by Peter T. Geach. In P. T. Geach and M. Black, editors, *Translations from the Philosophical Writings of Gottlob Frege*, pages 21–41. Basil Blackwell, 1960.

J. Gibbons. Calculating functional programs. In *Proceedings of ISRG/SERG Research Colloquium*. Oxford Brookes University, November 1997.

J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? In A. Corradini, M. Lenisa, and U. Montanari, editors, *Workshop on Coalgebraic Methods in Computer Science*, number 44.1 in Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, April 2001.

D. Gries and F. B. Schneider. Calculational logic. https://www.cs.cornell.edu/gries/Logic/intro.html, 2003.

D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer, October 22, 1993.

R. Hinze. La tour d'Hanoï. In A. Tolmach, editor, *International Conference on Functional Programming*, pages 3–10. ACM Press, 2009.

C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961a.

C. A. R. Hoare. Algorithm 63: Partition. *Communications of the ACM*, 4(7):321, 1961b.

C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961c.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, January 1962.

Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In S. L. Peyton Jones and M. Tofte, editors, *International Conference on Functional Programming*, volume 32, pages 164–175. ACM Press, 1997.

P. Hudak, R. J. M. Hughes, S. L. Peyton Jones, and P. L. Wadler. A history of Haskell: being lazy with class. In B. Ryder and B. Hailpern, editors, *History of Programming Languages III*, pages 1–55. ACM Press, 2007.

R. J. M. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22:141–144, 1986.

G. Hutton. *Programming in Haskell, 2nd Edition*. Cambridge University Press, 2016.

A. Kaldewaij. *Programming: the Derivation of Algorithms*. Prentice Hall, 1990.

Z. Manna and J. McCarthy. Properties of programs and partial function logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 27–37. Edinburgh University Press, 1969.

Z. Manna and A. Pnueli. Formalization of properties of functional programs. *Journal of the ACM*, 17(3):555–569, 1970.

J. Mazur. *Enlightening Symbols: A Short History of Mathematical Notation and Its Hidden Powers*. Princeton University Press, 2014. 中譯本：啟蒙的符號：數學符號的誕生、演化和隱藏的力量。譯者：洪萬生, 洪贊天, 英家銘, 黃俊瑋, 黃美倫, 鄭宜瑾。臉譜出版社，2015.

J. Misra. A visionary decision. In M. Broy, editor, *Constructive Methods in Computing Science: International Summer School directed by F.L. Bauer, M. Broy, E.W. Dijkstra, C.A.R. Hoare*, pages 1–3. Springer-Verlag, 1989.

S.-C. Mu. Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In R. Glück and O. de Moor, editors, *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 31–39, San Francisco, California, USA, jan 2008. ACM Press.

S.-C. Mu and R. S. Bird. Theory and applications of inverting functions as folds. *Science of Computer Programming (Special Issue for Mathematics of Program Construction)*, 51:87–116, 2003.

C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

G. O'Toole. *Hemingway Didn't Say That: The Truth Behind Familiar Quotations*, chapter With great power comes great responsibility. Little A, 2017. Published online on Quote Investigator: https://quoteinvestigator.com/2015/07/23/great-power/.

N. Sankar, L. Allis, J. Seldi, J. Camp, S. Kahr, G. Leave, R. Botti, A. Dill, D. A. Turner, and J. Picton-Warlow. Currying, or schonfinkeling? http://computer-programming-forum.com/23-functional/976f118bb90d8b15.htm, May 1997.

M. Schönfinkel. Uběr die bausteinen der mathematische logik. *Mathematische Annalen*, 92:305–316, 1924. Translated by Stefan Bauer-Mengelberg as "On the building blocks of mathematical logic" in Jean van Heijenoort, 1967. *A Source Book in Mathematical Logic*, 1879–1931.

C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 1967.

J. L. van de Snepscheut. *What Computing Is All About*. Springer, 1993.

P. L. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Marktoberdorf Summer School*, pages 233–264. Springer-Verlag, 1992.

H. Zantema. Longest segment problems. *Science of Computer Programming*, 18(1): 39–66, 1992.

APPENDIX

習題解答

Answer (習題 1.1) —

$$
\begin{aligned}
&myeven :: \mathsf{Int} \to \mathsf{Bool} \\
&myeven\ x = x\ `mod`\ 2 == 0\ .
\end{aligned}
$$

Answer (習題 1.2) —

$$
\begin{aligned}
&area\ \ :: \mathsf{Float} \to \mathsf{Float} \\
&area\ r = (22 / 7) \times r \times r\ .
\end{aligned}
$$

Answer (習題 1.6) — 根據外延相等，我們需證明 $(\forall x \cdot id \cdot f = f = f \cdot id)$. 推論如下：

$$
\begin{aligned}
&(id \cdot f)\ x \\
=\ &\quad \{\ (\cdot)\ 之定義\ \} \\
&id\ (f\ x) \\
=\ &\quad \{\ id\ 之定義\ \} \\
&f\ x \\
=\ &\quad \{\ id\ 之定義\ \} \\
&f\ (id\ x) \\
=\ &\quad \{\ (\cdot)\ 之定義\ \} \\
&(f \cdot id)\ x\ .
\end{aligned}
$$

Answer (習題 1.10) — 欲證明 $curry \cdot uncurry = id$：

$$
\begin{aligned}
& curry \cdot uncurry = id \\
\equiv \quad & \{\text{外延相等及 } id \text{ 之定義}\} \\
& (\forall f : curry\ (uncurry\ f) = f) \\
\equiv \quad & \{\text{外延相等}\} \\
& (\forall f\ x\ y : curry\ (uncurry\ f)\ x\ y = f\ x\ y)\ .
\end{aligned}
$$

因此我們證明 $curry\ (uncurry\ f)\ x\ y = f\ x\ y$ 如下：

$$
\begin{aligned}
& curry\ (uncurry\ f)\ x\ y \\
= \quad & \{curry \text{ 之定義}\} \\
& uncurry\ f\ (x,y) \\
= \quad & \{uncurry \text{ 之定義}\} \\
& f\ x\ y\ .
\end{aligned}
$$

與此相似，欲證明 $uncurry \cdot curry = id$ 我們須證明 $uncurry\ (curry\ f)\ (x,y) = f\ (x,y)$，其證明也與上面的證明類似。

Answer (習題 1.11) — $map\ (uncurry\ (\$)) :: \mathsf{List}\ ((a \to b) \times a) \to \mathsf{List}\ b$. 它拿一個內容均為「(函數 $\times$ 參數)」序對的串列，將每個函數作用在其參數上。例如：

$$map\ (uncurry\ (\$))\ [((1+),3),(square,4),(smaller\ 5,3)]$$

會得到 $[1+3, 4\times 4, 3]$.

Answer (習題 1.12) —

$$
\begin{aligned}
& tails :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\
& tails\ xs = map\ (\lambda n \to drop\ n\ xs)\ [0\,..\,length\ xs]\ .
\end{aligned}
$$

Answer (習題 1.13) —

$$
\begin{aligned}
& squaresUpTo :: \mathsf{Int} \to \mathsf{List}\ \mathsf{Int} \\
& squaresUpTo\ n = takeWhile\ (\leqslant n)\ (map\ square\ [0\,..])\ .
\end{aligned}
$$

Answer (習題 1.14) — **Answer (習題 1.14)** — $zip = zipWith\ (,)$, 或 $zip = zipWith\ (\lambda x\ y \to (x,y))$.

Answer (習題 2.5) —

$$
\begin{aligned}
& reverse :: \mathsf{List}\ a \to \mathsf{List}\ a \\
& reverse\ [\,] \quad\ = [\,] \\
& reverse\ (x : xs) = reverse\ xs +\!\!+ [x]\ .
\end{aligned}
$$

另，關於 *reverse* 效率的討論詳見第 5.2 節。

Answer (習題 2.7) — 欲證明 $sum\ (map\ (x\times)\ ys) = x \times sum\ ys$, 在 ys 上歸納。
情況 $ys := [\,]$, 兩邊都歸約為 0.
情況 $ys := y : ys$:

$$
\begin{aligned}
& sum\ (map\ (x\times)\ (y:ys)) \\
= \quad & \{\ map\ 之定義\ \} \\
& sum\ (x \times y : map\ (x\times)\ ys) \\
= \quad & \{\ sum\ 之定義\ \} \\
& x \times y + sum\ (map\ (x\times)\ ys) \\
= \quad & \{\ 歸納假設\ \} \\
& x \times y + x \times sum\ ys \\
= \quad & \{\ 乘法與加法之分配律\ \} \\
& x \times (y + sum\ ys) \\
= \quad & \{\ sum\ 之定義\ \} \\
& x \times sum\ (y:ys)\ \ .
\end{aligned}
$$

Answer (習題 2.10) — 檢視 $length$, $(+\!\!+)$, 與 $(+)$ 的定義，會發現等號兩邊都須對 xs 做分析才能化簡。因此我們對 xs 做歸納。
狀況 $xs := [\,]$.

$$
\begin{aligned}
& length\ ([\,] +\!\!+ ys) \\
= \quad & \{\ (+\!\!+)\ 之定義\ \} \\
& length\ ys \\
= \quad & \{\ (+)\ 之定義\ \} \\
& \mathbf{0} + length\ ys \\
= \quad & \{\ length\ 之定義\ \} \\
& length\ [\,] + length\ ys\ \ .
\end{aligned}
$$

狀況 $xs := x : xs$.

$$
\begin{aligned}
& length\ ((x:xs) +\!\!+ ys) \\
= \quad & \{\ (+\!\!+)\ 之定義\ \} \\
& length\ (x : (xs +\!\!+ ys)) \\
= \quad & \{\ length\ 之定義\ \} \\
& \mathbf{1}_+\ (length\ (xs +\!\!+ ys)) \\
= \quad & \{\ 歸納假設\ \} \\
& \mathbf{1}_+\ (length\ xs + length\ ys) \\
= \quad & \{\ (+)\ 之定義\ \} \\
& (\mathbf{1}_+\ (length\ xs)) + length\ ys \\
= \quad & \{\ length\ 之定義\ \} \\
& length\ (x:xs) + length\ ys\ \ .
\end{aligned}
$$

Answer (習題 2.12) — 考慮 $xs : xss$ 的情況：

$$
\begin{aligned}
& length\ (concat\ (xs:xss)) \\
= \ & length\ (xs +\!\!+ concat\ xss)
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\text{因 } length\ (xs +\!\!+ ys) = length\ xs + length\ ys\} \\
&\quad length\ xs + length\ (concat\ xss) \\
&= \quad \{\text{歸納假設}\} \\
&\quad length\ xs + sum\ (map\ length\ xss) \\
&= sum\ (length\ xs : map\ length\ xss) \\
&= sum\ (map\ length\ (xs : xss)) \quad .
\end{aligned}
$$

我們需要的性質是 $length\ (xs +\!\!+ ys) = length\ xs + length\ ys$.

Answer (習題 2.13) — 考慮 $xs : xss$ 的情況：

$$
\begin{aligned}
&\quad map\ f\ (concat\ (xs : xss)) \\
&= map\ f\ (xs +\!\!+ concat\ xss) \\
&= \quad \{\text{因 } map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys\} \\
&\quad map\ f\ xs +\!\!+ map\ f\ (concat\ xss) \\
&= \quad \{\text{歸納假設}\} \\
&\quad map\ f\ xs +\!\!+ concat\ (map\ (map\ f\ xss)) \\
&= concat\ (map\ f\ xs : map\ (map\ f\ xss)) \\
&= concat\ (map\ (map\ f\ (xs : xss))) \quad .
\end{aligned}
$$

我們需要的性質是 $map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys$.

Answer (習題 2.19) — 在 xs 上做歸納。

$$
\begin{aligned}
&\quad all\ (\in (x : xs))\ (filter\ p\ (x : xs)) \\
&\equiv all\ (\in (x : xs))\ (\textbf{if } p\ x \textbf{ then } x : filter\ p\ xs \textbf{ else } filter\ p\ xs) \\
&\equiv \textbf{if } p\ x \textbf{ then } all\ (\in (x : xs))\ (x : filter\ p\ xs) \\
&\qquad\quad \textbf{else } all\ (\in (x : xs))\ (filter\ p\ xs) \\
&\equiv \textbf{if } p\ x \textbf{ then } x \in (x : xs) \wedge all\ (\in (x : xs))\ (filter\ p\ xs) \\
&\qquad\quad \textbf{else } all\ (\in (x : xs))\ (filter\ p\ xs) \\
&\Leftarrow \textbf{if } p\ x \textbf{ then } all\ (\in (x : xs))\ (filter\ p\ xs) \\
&\qquad\quad \textbf{else } all\ (\in (x : xs))\ (filter\ p\ xs) \\
&\equiv \quad \{(2.4)\} \\
&\quad all\ (\in (x : xs))\ (filter\ p\ xs) \\
&\Leftarrow \quad \{\text{因習題 2.17, } z \in (x : xs) \Leftarrow z \in xs\} \\
&\quad all\ (\in xs)\ (filter\ p\ xs) \\
&\Leftarrow \mathsf{True} \quad .
\end{aligned}
$$

Answer (習題 2.20) —

$$
\begin{aligned}
&inits^{+} :: \mathsf{List}\ a \to \mathsf{List}\ (\mathsf{List}\ a) \\
&inits^{+}\ [\,] \quad\quad = [\,] \\
&inits^{+}\ (x : xs) = [x] : map\ (x:)\ (inits^{+}\ xs) \quad .
\end{aligned}
$$

Answer (習題 2.21) — 在 xs 上做歸納。
情況 $xs := []$. 此時等號兩邊都是 $[[]]$.
情況 $xs := x : xs$.

$$
\begin{aligned}
& map\ (\lambda n \to take\ n\ (x:xs))\ (upto\ (length\ (x:xs))) \\
= \quad & \{\ length\ 之定義\ \} \\
& map\ (\lambda n \to take\ n\ (x:xs))\ (upto\ (\mathbf{1}_+\ (length\ xs))) \\
= \quad & \{\ upto\ 之定義\ \} \\
& map\ (\lambda n \to take\ n\ (x:xs))\ (0 : map\ (\mathbf{1}_+)\ (upto\ (length\ xs))) \\
= \quad & \{\ length\ 之定義\ \} \\
& map\ (\lambda n \to take\ n\ (x:xs))\ (0 : map\ (\mathbf{1}_+)\ (upto\ (length\ xs))) \\
= \quad & \{\ map之定義，take\ 0\ (x:xs) = []\ \} \\
& []\ :\ map\ (\lambda n \to take\ n\ (x:xs))\ (map\ (\mathbf{1}_+)\ (upto\ (length\ xs))) \\
= \quad & \{\ 定理\ 2.2：map\ f \cdot map\ g = map\ (f \cdot g)\ \} \\
& []\ :\ map\ (\lambda n \to take\ (\mathbf{1}_+\ n)\ (x:xs))\ (upto\ (length\ xs)) \\
= \quad & \{\ take\ 之定義\ \} \\
& []\ :\ map\ (\lambda n \to x : take\ n\ xs)\ (upto\ (length\ xs)) \\
= \quad & \{\ 定理\ 2.2：map\ f \cdot map\ g = map\ (f \cdot g)\ \} \\
& []\ :\ map\ (x:)\ (map\ (\lambda n \to take\ n\ xs)\ (upto\ (length\ xs))) \\
= \quad & \{\ 歸納假設\ \} \\
& []\ :\ map\ (x:)\ (inits\ xs) \\
= \quad & \{\ inits\ 之定義\ \} \\
& inits\ (x:xs)\ \ .
\end{aligned}
$$

Answer (習題 2.22) — 考慮如何從 $segments\ [1,2,3]$ 湊出 $segments\ [0,1,2,3]$.
前者應當有的結果可能是：

$$[[],[1],[1,2],[1,2,3],[2],[2,3],[3]]\ \ .$$

而 $segments\ [0,1,2,3]$ 的結果可能是：

$$[[],[0],[0,1],[0,1,2],[0,1,2,3],[1],[1,2],[1,2,3],[2],[2,3],[3]]\ \ .$$

乍看之下，要多做的一步是：在空串列或所有 1開頭的串列前面補上 0. 之所以只擴充 1開頭的串列，因為只有這些是原本便靠在左邊，和 0 相鄰的（也就是說它們是 $[1,2,3]$的前段）。
然而，當輸入串列型別為 List a（而不是特定的 List Int, List Char 時），我們無法用比較的方式找出這些前段。函數 $segments$ 傳回的資訊不夠多。如果要歸納定義，我們必須把「前段」們和其他的區段分開。

Answer (習題 2.25) — 在 xs 上做歸納。基底狀況很容易成立。考慮 $xs := y : xs$:

$$
\begin{aligned}
& length\ (fan\ x\ (y:xs)) \\
= \ & length\ ((x:y:xs) : map\ (y:)\ (fan\ x\ xs))
\end{aligned}
$$

$$
\begin{aligned}
&= \mathbf{1}_+ \ (length \ (map \ (y:) \ (fan \ x \ xs))) \\
&= \quad \{ \text{因 } length \cdot map \ f = length \text{ (練習 2.6)} \} \\
&\quad \mathbf{1}_+ \ (length \ (fan \ x \ xs)) \\
&= \quad \{ \text{歸納假設} \} \\
&\quad \mathbf{1}_+ \ (\mathbf{1}_+ \ (length \ xs)) \\
&= \mathbf{1}_+ \ (length \ (y:xs)) \ .
\end{aligned}
$$

Answer (習題 2.26) — 欲證明 $map \ length \ (perms \ xs) = map \ (const \ (length \ xs)) \ (perms \ xs)$, 在 xs 之上做歸納。基底狀況 $xs := [\,]$ 中，等號兩邊都化約為 $[0]$. 考慮歸納步驟 $xs := x : xs$:

$$
\begin{aligned}
&\quad map \ length \ (perms \ (x:xs)) \\
&= \quad \{ perms \text{ 之定義} \} \\
&\quad map \ length \cdot concat \cdot map \ (fan \ x) \cdot perms \ \$ \ xs \\
&= \quad \{ map \ f \cdot concat = concat \cdot map \ (map \ f) \text{ (習題 2.13), } map \text{ 融合} \} \\
&\quad concat \cdot map \ (map \ length \cdot fan \ x) \cdot perms \ \$ \ xs \\
&= \quad \{ map \ length \cdot fan \ x = (\lambda n \to repeatN \ n \ n) \cdot length \} \\
&\quad concat \cdot map \ ((\lambda n \to repeatN \ n \ n) \cdot (\mathbf{1}_+) \cdot length) \cdot perms \ \$ \ xs \\
&= \quad \{ map \text{ 融合} \} \\
&\quad concat \cdot map \ ((\lambda n \to repeatN \ n \ n) \cdot (\mathbf{1}_+)) \cdot map \ length \cdot perms \ \$ \ xs \\
&= \quad \{ \text{歸納假設} \} \\
&\quad concat \cdot map \ ((\lambda n \to repeatN \ n \ n) \cdot (\mathbf{1}_+)) \cdot map \ (const \ (length \ xs)) \cdot perms \ \$ \ xs \\
&= \quad \{ map \text{ 融合} \} \\
&\quad concat \cdot map \ ((\lambda n \to repeatN \ n \ n) \cdot (\mathbf{1}_+) \cdot const \ (length \ xs)) \cdot perms \ \$ \ xs \\
&= \quad \{ \text{歸約 } \lambda \text{ 算式} \} \\
&\quad concat \cdot map \ (repeatN \ (\mathbf{1}_+ \ (length \ xs)) \cdot (\mathbf{1}_+) \cdot length) \cdot perms \ \$ \ xs \\
&= \quad \{ length \cdot fan \ x = (\mathbf{1}_+) \cdot length \text{ (習題 2.25)} \} \\
&\quad concat \cdot map \ (repeatN \ (\mathbf{1}_+ \ (length \ xs)) \cdot length \cdot fan \ x) \cdot perms \ \$ \ xs \\
&= \quad \{ \text{[todo: ???] } map \ (const \ y) = repeatN \ y \cdot length \} \\
&\quad concat \cdot map \ (map \ (const \ (\mathbf{1}_+ \ (length \ xs))) \cdot fan \ x) \cdot perms \ \$ \ xs \\
&= \quad \{ map \ f \cdot concat = concat \cdot map \ (map \ f) \text{ (習題 2.13)} \} \\
&\quad map \ (const \ (length \ (x:xs))) \cdot concat \cdot map \ (fan \ x) \cdot perms \ \$ \ xs \\
&= \quad \{ perms \text{ 之定義} \} \\
&\quad map \ (const \ (length \ (x:xs))) \ (perms \ (x:xs)) \ .
\end{aligned}
$$

Answer (習題 2.27) —

$$
\begin{aligned}
&splits :: \mathsf{List} \ a \to \mathsf{List} \ (\mathsf{List} \ a \times \mathsf{List} \ a) \\
&splits \ [\,] \quad\quad = [([\,],[\,])] \\
&splits \ (x:xs) = ([\,], x:xs) : map \ ((x:) \times id) \ (splits \ xs) \ .
\end{aligned}
$$

其中 $(f \times g) \ (x,y) = (f \ x, g \ y)$.

Answer (習題 2.28) — 將式子改寫為 $length\ (sublists\ xs) = exp\ 2\ (length\ xs)$, 在 xs 上做歸納。歸納步驟 $xs := x : xs$ 為：

$$
\begin{aligned}
& length\ (sublists\ (x : xs)) \\
= {} & length\ (sublists\ xs +\!\!+ map\ (x:)\ (sublists\ xs)) \\
= {} & \quad \{ \text{因 (2.1) } length\ (xs +\!\!+ ys) = length\ xs + length\ ys \} \\
& length\ (sublists\ xs) + length\ (map\ (x:)\ (sublists\ xs)) \\
= {} & \quad \{ \text{因 } length \cdot map = length \text{ (練習 2.6)} \} \\
& length\ (sublists\ xs) + length\ (sublists\ xs) \\
= {} & \quad \{ \text{因 } x + x = 2 \times x \} \\
& 2 \times length\ (sublists\ xs) \\
= {} & \quad \{ \text{歸納步驟} \} \\
& 2 \times exp\ 2\ (length\ xs) \\
= {} & exp\ 2\ (\mathbf{1}_{+}\ (length\ xs)) \\
= {} & exp\ 2\ (length\ (x : xs)) \quad .
\end{aligned}
$$

Answer (習題 2.29) —

$$
\begin{aligned}
& ((y:) \cdot (ys +\!\!+))\ xs \\
= {} & \quad \{ \text{definition of } (\cdot) \} \\
& y : (ys +\!\!+ xs) \\
= {} & \quad \{ \text{definition of } (+\!\!+) \} \\
& (y : ys) +\!\!+ xs \quad .
\end{aligned}
$$

Answer (習題 2.30) —

$$
\begin{aligned}
& ((+\!\!+ zs) \cdot (y:))\ xs \\
= {} & \quad \{ \text{definition of } (\cdot) \} \\
& (+\!\!+ zs)\ (y : xs) \\
= {} & (y : xs) +\!\!+ zs \\
= {} & \quad \{ \text{definition of } (+\!\!+) \} \\
& y : (xs +\!\!+ zs) \\
= {} & (y:)\ ((+\!\!+ zs)\ xs) \\
= {} & \quad \{ \text{definition of } (\cdot) \} \\
& ((y:) \cdot (+\!\!+ zs))\ xs \quad .
\end{aligned}
$$

Answer (習題 2.31) —

ITree 上之歸納法： $(\forall t :: \mathsf{ITree}\ a \cdot \mathsf{P}\ t) \Leftarrow$
$(\mathsf{P}\ \mathsf{Null} \wedge (\forall x\ t\ u \cdot \mathsf{P}\ (\mathsf{Node}\ x\ t\ u) \Leftarrow \mathsf{P}\ t \wedge \mathsf{P}\ u)\ .$

Answer (習題 2.32) —

$$
\begin{aligned}
&\mathit{tags} :: \mathsf{ITree}\ a \to \mathsf{List}\ a \\
&\mathit{tags}\ \mathsf{Null} \qquad\quad = [\,] \\
&\mathit{tags}\ (\mathsf{Node}\ x\ t\ u) = \mathit{tags}\ t +\!\!+ [x] +\!\!+ \mathit{tags}\ u \quad .
\end{aligned}
$$

Answer (習題 2.33) —

$$
\begin{aligned}
&\mathit{size} :: \mathsf{ITree}\ a \to \mathbb{N} \\
&\mathit{size}\ \mathsf{Null} \qquad\quad = \mathbf{0} \\
&\mathit{size}\ (\mathsf{Node}\ x\ t\ u) = \mathbf{1}_+\ (\mathit{size}\ t + \mathit{size}\ u) \quad .
\end{aligned}
$$

Answer (習題 2.34) — 在 t 之上做歸納。當 $t := \mathsf{Null}$，顯然等號兩邊都歸約為 0. 當 $t := \mathsf{Node}\ x\ t\ u$:

$$
\begin{aligned}
& \mathit{length}\ (\mathit{tags}\ (\mathsf{Node}\ x\ t\ u)) \\
=\quad & \{\ \mathit{length} \text{ 與 } \mathit{leaves} \text{ 之定義}\ \} \\
& \mathbf{1}_+\ (\mathit{length}\ (\mathit{tags}\ t) + \mathit{length}\ (\mathit{tags}\ u)) \\
=\quad & \{\ \text{歸納假設}\ \} \\
& \mathbf{1}_+\ (\mathit{size}\ t + \mathit{size}\ u) \\
=\quad & \{\ \mathit{size} \text{ 之定義}\ \} \\
& \mathit{size}\ (\mathsf{Node}\ x\ t\ u) \quad .
\end{aligned}
$$

Answer (習題 2.37) — $\mathit{head}\ [\,] : \mathit{tail}\ [\,] \neq [\,]$.

Answer (習題 2.38) — 針對 m 做歸納。當 $m := \mathbf{0}$, 等號兩邊都歸約成 $[\,]$。考慮 $m := \mathbf{1}_+\ m$. 此時針對 xs 做分析。當 $xs := [\,]$, 等號兩邊亦都歸約成 $[\,]$。考慮 $xs := x : xs$ 的情況：

$$
\begin{aligned}
& \mathit{take}\ (\mathbf{1}_+\ m)\ (\mathit{take}\ (\mathbf{1}_+\ m + n)\ (x : xs)) \\
=\quad & \{\ (+) \text{ 與 } \mathit{take} \text{ 之定義}\ \} \\
& \mathit{take}\ (\mathbf{1}_+\ m)\ (x : \mathit{take}\ (m + n)\ xs) \\
=\quad & \{\ \mathit{take} \text{ 之定義}\ \} \\
& x : \mathit{take}\ m\ (\mathit{take}\ (m + n)\ xs) \\
=\quad & \{\ \text{歸納假設}\ \} \\
& x : \mathit{take}\ m\ xs \\
=\quad & \{\ \mathit{take} \text{ 之定義}\ \} \\
& \mathit{take}\ (\mathbf{1}_+\ m)\ xs \quad .
\end{aligned}
$$

Answer (習題 2.39) — 針對 m 做歸納。當 $m := \mathbf{0}$, 等號兩邊都歸約成 $\mathit{take}\ n\ xs$。考慮 $m := \mathbf{1}_+\ m$. 此時針對 xs 做分析。當 $xs := [\,]$, 等號兩邊亦都歸約成 $[\,]$。考慮 $xs := x : xs$ 的情況：

$$
\begin{aligned}
& \mathit{drop}\ (\mathbf{1}_+\ m)\ (\mathit{take}\ (\mathbf{1}_+\ m + n)\ (x : xs)) \\
=\quad & \{\ (+) \text{ 與 } \mathit{take} \text{ 之定義}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& drop\ (\mathbf{1}_+\ m)\ (x : take\ (m+n)\ xs) \\
= & \quad \{\ take\ 之定義\ \} \\
& drop\ n\ (take\ (m+n)\ xs) \\
= & \quad \{\ 歸納假設\ \} \\
& take\ n\ (drop\ m\ xs) \\
= & \quad \{\ take\ 之定義\ \} \\
& take\ n\ (drop\ (\mathbf{1}_+\ m)\ (x : xs))\ .
\end{aligned}
$$

Answer (習題 2.40) — 針對 m 做歸納。當 $m := \mathbf{0}$, 等號兩邊都歸約成 $drop\ n\ xs$。考慮 $m := \mathbf{1}_+\ m$. 此時針對 xs 做分析。當 $xs := [\,]$, 等號兩邊亦都歸約成 $[\,]$。考慮 $xs := x : xs$ 的情況：

$$
\begin{aligned}
& drop\ (\mathbf{1}_+\ m+n)\ (x : xs) \\
= & \quad \{\ (+)\ 與\ take\ 之定義\ \} \\
& drop\ (m+n)\ xs \\
= & \quad \{\ 歸納假設\ \} \\
& drop\ n\ (drop\ m\ xs) \\
= & \quad \{\ drop\ 之定義\ \} \\
& drop\ n\ (drop\ (\mathbf{1}_+\ m)\ (x : xs))\ .
\end{aligned}
$$

Answer (習題 2.42) — 使用完全歸納。
狀況 $n := \mathbf{0}$: 等號兩邊都歸約為 $[\,]$.
狀況 $n > \mathbf{0}$。假設對於任何 $0 \leqslant i < n$, $sum\ (binary\ i) = i$.

$$
\begin{aligned}
& sum\ (binary\ n) = \\
= & \quad \{\ 令\ m = last\ (takeWhile\ (\leqslant n)\ twos),\ sum\ 之定義\ \} \\
& m + sum\ (binary\ (n-m)) \\
= & \quad \{\ 因\ n-m < n,\ 歸納假設\ \} \\
& m + (n-m) \\
= & n\ .
\end{aligned}
$$

Answer (習題 2.43) — 本證明的關鍵性質是 $\alpha^2 = (3+\sqrt{5})/2 = \alpha + 1$. 使用完全歸納證明 $fib\ (2+n) > \alpha^n$，以 $n := 1$ 和 $n := 2$ 當基底狀況。
狀況 $n := 1$: $fib\ 3 = 2 > \alpha$.
狀況 $n := 2$: $fib\ 4 = 3 > (3+\sqrt{5})/2 = \alpha^2$.
狀況 $n := 2+n$ 且 $2+n > 4$。假設對於任何 $3 \leqslant i < 2+n$, $fib\ i > \alpha^{i-2}$. 論證：

$$
\begin{aligned}
& fib\ (2+n) \\
= & fib\ (1+n) + fib\ n \\
> & \quad \{\ 歸納假設\ \} \\
& \alpha^{n-1} + \alpha^{n-2} \\
= & \quad \{\ 提出\ \alpha^{n-2}\ \} \\
& (\alpha+1) \times \alpha^{n-2}
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\ \alpha^2 = \alpha + 1\ \} \\
&\quad \alpha^2 \times \alpha^{n-2} \\
&= \alpha^n \quad .
\end{aligned}
$$

Answer (習題 2.44) — 在 xs 上做歸納。

$$
\begin{aligned}
&\quad all\ ((x : xs)\ ==)\ (zipWith\ (+\!\!+)\ (inits\ (x : xs))\ (tails\ (x : xs))) \\
&\equiv all\ ((x : xs)\ ==)\ (zipWith\ (+\!\!+)\ ([\,] : map\ (x :)\ (inits\ xs))\ ((x : xs) : tails\ xs)) \\
&\equiv all\ ((x : xs)\ ==)\ (([\,] +\!\!+ (x : xs)) : zipWith\ (+\!\!+)\ (map\ (x :)\ (inits\ xs))\ (tails\ xs)) \\
&\equiv ((x : xs) == (x : xs)) \wedge \\
&\quad all\ ((x : xs)\ ==)\ (zipWith\ (+\!\!+)\ (map\ (x :)\ (inits\ xs))\ (tails\ xs))
\end{aligned}
$$

Answer (習題 3.2) — 在 t 之上做歸納。基底情況 $t := \mathsf{E}$ 很容易建立。以下只示範一種歸納情況。
狀況 $t := \mathsf{B}\ t\ x\ u$, $k < x$:

$$
\begin{aligned}
&\quad balanced\ (ins\ k\ (\mathsf{B}\ t\ x\ u)) \\
&= \quad \{\ ins\ 之定義；k < x\ \} \\
&\quad balanced\ (rotate\ (ins\ k\ t)\ x\ u) \\
&\Leftarrow \quad \{\ 引理\ 3.6\ \} \\
&\quad balanced\ (ins\ k\ t) \wedge balanced\ u \wedge bheight\ (ins\ k\ t) = bheight\ u \\
&= \quad \{\ 定理\ 3.2\ \} \\
&\quad balanced\ (ins\ k\ t) \wedge balanced\ u \wedge bheight\ t = bheight\ u \\
&\Leftarrow \quad \{\ 歸納假設\ \} \\
&\quad balanced\ t \wedge balanced\ u \wedge bheight\ t = bheight\ u \\
&= \quad \{\ balanced\ 之定義\ \} \\
&\quad balanced\ (\mathsf{B}\ t\ x\ u) \quad .
\end{aligned}
$$

Answer (習題 3.3) — 以下只示範其中一種狀況。
狀況: $(t, x, u) := (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v, z, w)$.

$$
\begin{aligned}
&\quad balanced\ (rotate\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v)\ z\ w) \\
&= \quad \{\ rotate\ 之定義\ \} \\
&\quad balanced\ (\mathsf{R}\ (\mathsf{B}\ t\ x\ u)\ y\ (\mathsf{B}\ v\ z\ w)) \\
&= \quad \{\ balanced\ 之定義\ \} \\
&\quad bheight\ (\mathsf{B}\ t\ x\ u) = bheight\ (\mathsf{B}\ v\ z\ w) \wedge \\
&\quad balanced\ (\mathsf{B}\ t\ x\ u) \wedge balanced\ (\mathsf{B}\ v\ z\ w) \\
&= \quad \{\ bheight\ 之定義\ \} \\
&\quad 1 + (bheight\ t \uparrow bheight\ u) = 1 + (bheight\ v \uparrow beight\ w) \wedge \\
&\quad bheight\ t = bheight\ u \wedge balanced\ t \wedge balanced\ u \wedge \\
&\quad bheight\ v = bheight\ w \wedge balanced\ v \wedge balanced\ w \\
&\Leftarrow \quad \{\ balanced\ 與\ bheight\ 之定義；算數性質\ \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{bheight}\ (\mathsf{R}\ t\ x\ u) = \mathit{bheight}\ v \wedge \\
& \mathit{balanced}\ (\mathsf{R}\ t\ x\ u) \wedge \mathit{balanced}\ v \wedge \mathit{balanced}\ w \wedge \\
& \mathit{bheight}\ t \uparrow \mathit{bheight}\ u \uparrow \mathit{bheight}\ v = \mathit{bheight}\ w \\
= \quad & \{\ \mathit{balanced}\ \text{與}\ \mathit{bheight}\ \text{之定義}\ \} \\
& \mathit{balanced}\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v) \wedge \mathit{balanced}\ w \wedge \\
& \quad \mathit{bheight}\ \mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v = \mathit{bheight}\ w \quad .
\end{aligned}
$$

Answer (習題 3.4) — 本證明只是一一檢查每個狀況。以 1. $(\mathit{infrared}\ t \vee \mathit{semiRB}\ t) \wedge \mathit{semiRB}\ u \Rightarrow \mathit{semiRB}\ (\mathit{rotate}\ t\ x\ u)$ 為例，由於有 $\mathit{semiRB}\ u$, 我們只需檢查 rotate 的第一、第二、與最後一個狀況。以第一個狀況為例：
狀況: $(t,x,u) := (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v, z, w)$:

$$
\begin{aligned}
& \mathit{semiRB}\ (\mathit{rotate}\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v)\ z\ w) \\
= \quad & \{\ \mathit{rotate}\ \text{之定義}\ \} \\
& \mathit{semiRB}\ (\mathsf{R}\ (\mathsf{B}\ t\ x\ u)\ y\ (\mathsf{B}\ v\ z\ w)) \\
= \quad & \{\ \mathit{semiRB}\ \text{與}\ \mathit{color}\ \text{之定義}\ \} \\
& \mathit{color}\ t = \mathit{color}\ u = \mathit{color}\ v = \mathit{color}\ w = \mathsf{Blk} \wedge \\
& \mathit{semiRB}\ t \wedge \mathit{semiRB}\ u \wedge \mathit{semiRB}\ v \wedge \mathit{semiRB}\ w \\
= \quad & \{\ \mathit{semiRB}\ \text{之定義}\ \} \\
& \mathit{color}\ v = \mathit{color}\ w = \mathsf{Blk} \wedge \\
& \mathit{semiRB}\ (\mathsf{R}\ t\ x\ u) \wedge \mathit{semiRB}\ v \wedge \mathit{semiRB}\ w \\
= \quad & \{\ \mathit{infrared}\ \text{之定義}\ \} \\
& \mathit{infrared}\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v) \wedge \mathit{semiRB}\ w \\
= \quad & \{\ \text{命題邏輯，}\ \mathit{semiRB}\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v) = \mathsf{False}\ \} \\
& (\mathit{infrared}\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v) \vee \mathit{semiRB}\ (\mathsf{R}\ (\mathsf{R}\ t\ x\ u)\ y\ v)) \wedge \mathit{semiRB}\ w \quad .
\end{aligned}
$$

Answer (習題 5.1) — 顯然 $\mathit{sum}\ (\mathit{descend}\ 0) = 0$. 考慮歸納情況：

$$
\begin{aligned}
& \mathit{sum}\ (\mathit{descend}\ (\mathbf{1}_+\ n)) \\
= \quad & \{\ \mathit{descend}\ \text{之定義}\ \} \\
& \mathit{sum}\ (\mathbf{1}_+\ n : \mathit{descend}\ n) \\
= \quad & \{\ \mathit{sum}\ \text{之定義}\ \} \\
& (\mathbf{1}_+\ n) + \mathit{sum}\ (\mathit{descend}\ n) \\
= \quad & \{\ \mathit{sumseries}\ \text{之定義}\ \} \\
& (\mathbf{1}_+\ n) + \mathit{sumseries}\ n \quad .
\end{aligned}
$$

因此

$$
\begin{aligned}
& \mathit{sumseries}\ \mathbf{0} \qquad = \mathbf{0} \\
& \mathit{sumseries}\ (\mathbf{1}_+\ n) = (\mathbf{1}_+\ n) + \mathit{sumseries}\ n \quad .
\end{aligned}
$$

Answer (習題 5.2) — 顯然 $repeatN\ (0,x) = [\,]$. 至於歸納情況，演算如下：

$$
\begin{aligned}
& repeatN\ (\mathbf{1}_+\ n,x) \\
= \quad & \{\ repeatN\ 之定義\ \} \\
& map\ (const\ x)\ (descend\ (\mathbf{1}_+\ n)) \\
= \quad & \{\ descend\ 之定義\ \} \\
& map\ (const\ x)\ (\mathbf{1}_+\ n : descend\ n) \\
= \quad & \{\ map\ 與\ const\ 之定義\ \} \\
& x : map\ (const\ x)\ (descend\ n) \\
= \quad & \{\ repeatN\ 之定義\ \} \\
& x : repeatN\ (n,x)\ \ .
\end{aligned}
$$

因此，

$$
\begin{aligned}
repeatN\ (\mathbf{0},\ \ \ x) &= [\,] \\
repeatN\ (\mathbf{1}_+\ n,x) &= x : repeatN\ (n,x)\ \ .
\end{aligned}
$$

Answer (習題 5.3) — 基底狀況：

$$
\begin{aligned}
& rld\ [\,] \\
= \quad & \{\ rld\ 之定義\ \} \\
& concat\ (map\ repeatN\ [\,]) \\
= \quad & \{\ map\ 與\ concat\ 之定義\ \} \\
& [\,]\ \ .
\end{aligned}
$$

歸納狀況：

$$
\begin{aligned}
& rld\ ((n,x) : xs) \\
= \quad & \{\ rld\ 之定義\ \} \\
& concat\ (map\ repeatN\ ((n,x) : xs)) \\
= \quad & \{\ map\ 之定義\ \} \\
& concat\ (repeatN\ (n,x) : map\ repeatN\ xs) \\
= \quad & \{\ concat\ 之定義\ \} \\
& repeatN\ (n,x) +\!\!+ concat\ (map\ repeatN\ xs) \\
= \quad & \{\ rld\ 之定義\ \} \\
& repeatN\ (n,x) +\!\!+ rld\ xs\ \ .
\end{aligned}
$$

因此我們已推導出：

$$
\begin{aligned}
rld\ [\,] &= [\,] \\
rld\ ((n,x) : xs) &= repeatN\ (n,x) +\!\!+ rld\ xs\ \ .
\end{aligned}
$$

Answer (習題 5.4) — 顯然 $select\ [\,] = [\,]$。考慮 $xs := x : xs$ 的情況：

$$
\begin{aligned}
& select\ (x:xs) \\
= \quad & \{\ select \text{ 與 } delete \text{ 之定義}\ \} \\
& zip\ (x:xs)\ (xs:map\ (x:)\ (delete\ xs)) \\
= \quad & \{\ zip \text{ 之定義}\ \} \\
& (x,xs):zip\ xs\ (map\ (x:)\ (delete\ xs)) \\
= \quad & \{\ zip\ xs\ (map\ f\ ys) = map\ (id \times f)\ (zip\ xs\ ys)\ \} \\
& (x,xs):map\ (id \times (x:))\ (zip\ xs\ (delete\ xs)) \\
= \quad & \{\ select \text{ 之定義}\ \} \\
& (x,xs):map\ (id \times (x:))\ (select\ xs)\ \ .
\end{aligned}
$$

因此，

$$
\begin{aligned}
select\ [\,] &= [\,] \\
select\ (x:xs) &= (x,xs):map\ (id \times (x:))\ (select\ xs)\ \ .
\end{aligned}
$$

Answer (習題 5.5) — 以下只列出歸納狀況：

$$
\begin{aligned}
& delete\ (x:xs) \\
= \quad & \{\ delete \text{ 之定義}\ \} \\
& map\ (del\ (x:xs))\ [0\,..\,length\ (x:xs)-1] \\
= \quad & \{\ length \text{ 之定義，簡單運算}\ \} \\
& map\ (del\ (x:xs))\ [0\,..\,length\ xs] \\
= \quad & \{\ \text{由於 } length\ xs \geqslant 0, \text{使用 (5.3)}\ \} \\
& map\ (del\ (x:xs))\ (0:map\ (1+)\ [0\,..\,length\ xs-1]) \\
= \quad & \{\ map \text{ 之定義}\ \} \\
& del\ (x:xs)\ 0:map\ (del\ (x:xs))\ (map\ (1+)\ [0\,..\,length\ xs-1]) \\
= \quad & \{\ map \text{ 融合}\ \} \\
& del\ (x:xs)\ 0:map\ (del\ (x:xs)\cdot(1+))\ [0\,..\,length\ xs-1]\ \ .
\end{aligned}
$$

在此暫停一下，觀察 $del\ (x:xs)$. 顯然, $del\ (x:xs)\ 0 = xs$. 至於 $del\ (x:xs)\cdot(1+)$，我們演算看看:

$$
\begin{aligned}
& (del\ (x:xs)\cdot(1+))\ i \\
= \quad & \{\ (\cdot) \text{ 與 } del \text{ 之定義}\ \} \\
& take\ (1+i)\ (x:xs) +\!\!+ drop\ (1+(1+i))\ (x:xs) \\
= \quad & \{\ take \text{ 與 } drop \text{ 之定義}\ \} \\
& x:take\ i\ xs +\!\!+ drop\ (1+i)\ xs \\
= \quad & \{\ del \text{ 之定義}\ \} \\
& x:del\ xs\ i \\
= \quad & \{\ (\cdot) \text{ 之定義}\ \} \\
& ((x:)\cdot del\ xs)\ i\ \ .
\end{aligned}
$$

繼續之前的演算：

$$
\begin{aligned}
& del\ (x:xs)\ 0:map\ (del\ (x:xs)\cdot(1+))\ [0\,..\,length\ xs-1] \\
= \quad & \{\ \text{前述之演算}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& xs : map\ ((x:) \cdot del\ xs)\ [0\,..\,length\ xs - 1] \\
= & \quad \{\ map\ 融合\ \} \\
& xs : map\ (x:)\ (map\ (del\ xs)\ [0\,..\,length\ xs - 1]) \\
= & \quad \{\ delete\ 之定義\ \} \\
& xs : map\ (x:)\ (delete\ xs)\ \ .
\end{aligned}
$$

由此，我們導出了正文中 *delete* 的歸納定義.

Answer (習題 5.7) — 令 t 為一個向右傾斜的樹：

$$t = \mathsf{Bin}\ (\mathsf{Tip}\ 1)\ (\mathsf{Bin}\ (\mathsf{Tip}\ 2)\,..\,(\mathsf{Bin}\ (\mathsf{Tip}\ (n-1))\ (\mathsf{Tip}\ n)))\ \ ,$$

$tip\ t$ 展開成為 $[1] +\!\!+ ([2]\,..\,([n-1] +\!\!+ [n]))$, 可在 $O(n)$ 時間內歸約成範式。這是最好的情況。令 u 為一個向左傾斜的樹：

$$
\begin{aligned}
u = \mathsf{Bin}\ & (\mathsf{Bin}\ (...(\mathsf{Bin}\ (\mathsf{Bin}\ (\mathsf{Tip}\ 1)\ (\mathsf{Tip}\ 2))\ (\mathsf{Tip}\ 3)))\,... \\
& (\mathsf{Tip}\ (n-1)))\ (\mathsf{Tip}\ n)\ \ ,
\end{aligned}
$$

$tip\ u$ 展開成為 $((..(([1] +\!\!+ [2]) +\!\!+ [3])\,..) .. +\!\!+ [n-1]) +\!\!+ [n]$, 需要 $O(n^2)$ 的時間，也是 *tips* 最壞情況的時間複雜度.

Answer (習題 5.8) — 欲證明 $zipWith\ (\otimes)\ as\ (map\ (\otimes x)\ bs) = map\ (\otimes x)\ (zipWith\ (\otimes)\ as\ bs)$, 在 as 上做歸納。當 $as := [\,]$, 等號兩邊都歸約成 $[\,]$. 考慮 $as := a : as$ 的情況。若 $bs := [\,]$, 等號兩邊仍均為 $[\,]$. 當 $bs := b : bs$:

$$
\begin{aligned}
& zipWith\ (\otimes)\ (a:as)\ (map\ (\otimes x)\ (b:bs)) \\
= & \quad \{\ zipWith\ 與\ map\ 之定義\ \} \\
& (a \otimes (b \otimes x)) : zipWith\ (\otimes)\ as\ (map\ (\otimes x)\ bs) \\
= & \quad \{\ 歸納假設\ \} \\
& (a \otimes (b \otimes x)) : map\ (\otimes x)\ (zipWith\ (\otimes)\ as\ bs) \\
= & \quad \{\ (\otimes)\ 滿足結合律\ \} \\
& ((a \otimes b) \otimes x) : map\ (\otimes x)\ (zipWith\ (\otimes)\ as\ bs) \\
= & \quad \{\ zipWith\ 與\ map\ 之定義\ \} \\
& map\ (\otimes x)\ (zipWith\ (\otimes)\ (a:as)\ (b:bs))\ \ .
\end{aligned}
$$

Answer (習題 5.11) —

$$
\begin{aligned}
& exp\ b\ 0 = 1 \\
& exp\ b\ n \mid even\ n = square\ (exp\ b\ (n\ `div`\ 2)) \\
& \qquad\quad \mid odd\ n\ \ = b \times square\ (exp\ b\ (n\ `div`\ 2))\ \ .
\end{aligned}
$$

由於遞迴呼叫中的 n 總是變小，本定義可視為 n 之上的良基歸納。

Answer (習題 5.12) — 試著展開-收回 $index\ (x:xs)$：

$$
\begin{aligned}
& index\ (x:xs) \\
= & \ zip\ [\mathbf{0}\,..\,]\ (x:xs)
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\, [\mathbf{0}..] = \mathbf{0} : [1..],\ zip \text{ 之定義} \,\} \\
&\quad (\mathbf{0}, x) : zip\ [1..]\ xs \quad .
\end{aligned}
$$

我們又發現 zip [1..] 無法收回成 index xs. 因此，我們定義：

$$
\begin{aligned}
&indexFrom :: \mathbb{N} \to \mathsf{List}\ a \to \mathsf{List}\ (\mathbb{N} \times a) \\
&indexFrom\ i = zip\ [i..] \quad .
\end{aligned}
$$

試著推導 $indexFrom$ 的歸納定義如下：

$$
\begin{aligned}
&\quad indexFrom\ i\ (x : xs) \\
&= zip\ [i..]\ (x : xs) \\
&= \quad \{\, [i..] = i : [\mathbf{1}_+\ i..],\ zip \text{ 之定義} \,\} \\
&\quad (i, x) : zip\ [\mathbf{1}_+\ i..]\ xs \\
&= (i, x) : indexFrom\ (\mathbf{1}_+\ i)\ xs \quad .
\end{aligned}
$$

因此我們已得到：

$$
\begin{aligned}
&indexFrom\ i\ [\,] \qquad = [\,] \\
&indexFrom\ i\ (x : xs) = i, x) : indexFrom\ (\mathbf{1}_+\ i)\ xs \quad .
\end{aligned}
$$

Answer (習題 5.13) — 考慮如下的定義：

$$
\begin{aligned}
&baosum :: \mathsf{Tree\ Int} \to (\mathsf{Bool}, \mathsf{Int}) \\
&baosum = \langle baobab, sumT \rangle \quad .
\end{aligned}
$$

如果 $baosum$ 有時間效率為 $O(n)$ 的定義，我們可重定義 $baobab = fst \cdot baosum$. 當 $t := \mathsf{Null}$, 我們有 $baosum\ \mathsf{Null} = (\mathsf{True}, 0)$. 考慮 $t := \mathsf{Node}\ x\ t\ u$:

$$
\begin{aligned}
&\quad baosum\ (\mathsf{Node}\ x\ t\ u) \\
&= \quad \{\, baosum \text{ 之定義} \,\} \\
&\quad (baobab\ (\mathsf{Node}\ x\ t\ u), sumT\ (\mathsf{Node}\ x\ t\ u)) \\
&= \quad \{\, baobab \text{ 與 } sumT \text{ 之定義} \,\} \\
&\quad (baobab\ t \wedge baobab\ u \wedge x > (sumT\ t + sumT\ u), \\
&\qquad x + sumT\ t + sumT\ u) \\
&= \quad \{\, \text{引入區域變數} \,\} \\
&\quad \mathbf{let}\ (b, y) = (baobab\ t, sumT\ t) \\
&\qquad\ \ (c, z) = (baobab\ u, sumT\ u) \\
&\quad \mathbf{in}\ (b \wedge c \wedge x > (y + z), x + y + z) \\
&= \quad \{\, baosum \text{ 之定義} \,\} \\
&\quad \mathbf{let}\ (b, y) = baosum\ t \\
&\qquad\ \ (c, z) = baosum\ u \\
&\quad \mathbf{in}\ (b \wedge c \wedge x > (y + z), x + y + z) \quad .
\end{aligned}
$$

如此，我們已經導出：

$$
\begin{aligned}
&baosum\ \mathsf{Null} &&= (\mathsf{True}, 0)\\
&baosum\ (\mathsf{Node}\ x\ t\ u) &&=\\
&\quad \textbf{let}\ (b,y) = baosum\ t\\
&\qquad\ (c,z) = baosum\ u\\
&\quad \textbf{in}\ (b \wedge c \wedge x > (y+z), x+y+z)\ .
\end{aligned}
$$

Answer (習題 5.14) — 定義：

$$
\begin{aligned}
&dd :: \mathsf{ETree}\ a \to (\mathsf{List}\ a, \mathbb{N})\\
&dd = \langle deepest, depth \rangle\ .
\end{aligned}
$$

顯然 $dd\ (\mathsf{Tip}\ x) = ([x], 0)$. 以下我們考慮歸納情況時，用一個非標準的語法同時處理守衛算式的三個式子：

$$
\begin{aligned}
&\quad dd\ (\mathsf{Bin}\ t\ u)\\
&= \quad \{\ dd, deepest, \text{與 } depth \text{ 之定義}\ \}\\
&\quad ((m < n \to deepest\ u\\
&\qquad \mid m == n \to deepest\ t +\!\!+ deepest\ u\\
&\qquad \mid m > n \to deepest\ t), 1 + (m \uparrow n))\\
&\quad \textbf{where}\ (m,n) = (depth\ t, depth\ u)\\
&= \quad \{\ \text{取出 } deepest\ t \text{ 與 } deepest\ u\ \}\\
&\quad ((m < n \to ys\\
&\qquad \mid m == n \to xs +\!\!+ ys\\
&\qquad \mid m > n \to xs), 1 + (m \uparrow n))\\
&\quad \textbf{where}\ ((xs,m),(ys,n)) = ((deepest\ t, depth\ t), (deepest\ u, depth\ u))\\
&= \quad \{\ dd \text{ 之定義，函數分配進條件判斷}\ \}\\
&\quad (m < n \to (ys, 1+n)\\
&\qquad \mid m == n \to (xs +\!\!+ ys, 1+n)\\
&\qquad \mid m > n \to (xs, 1+m))\\
&\quad \textbf{where}\ ((xs,m),(ys,n)) = (dd\ t, dd\ u)\ .
\end{aligned}
$$

因此我們得到：

$$
\begin{aligned}
&dd\ (\mathsf{Tip}\ x) &&= ([x], 0)\\
&dd\ (\mathsf{Bin}\ t\ u) &&\mid m < n = (ys, 1+n)\\
&&&\mid m == n = (xs +\!\!+ ys, 1+n)\\
&&&\mid m > n = (xs, 1+m)\ ,\\
&\quad \textbf{where}\ ((xs,m),(ys,n)) = (dd\ t, dd\ u)\ .
\end{aligned}
$$

Answer (習題 5.15) — 定義：

$$
\begin{aligned}
&balHeight :: \mathsf{RBTree} \to (\mathsf{Bool}, \mathbb{N})\\
&balHeight = \langle balanced, bheight \rangle\ .
\end{aligned}
$$

我們有 $balanced = fst \cdot balHeight$. 顯然 $balHeight\ \mathsf{E} = (\mathsf{True}, 0)$. 考慮 $\mathsf{B}\ t\ x\ u$ 的狀況：

$$
\begin{aligned}
& balHeight\ (\mathsf{B}\ t\ x\ u) \\
= & \quad \{\ balHeight\ 之定義\ \} \\
& (balanced\ (\mathsf{B}\ t\ x\ u), bheight\ (\mathsf{B}\ t\ x\ u)) \\
= & \quad \{\ balanced\ 與\ bheight\ 之定義\ \} \\
& (bheight\ t == bheight\ u \wedge balanced\ t \wedge balanced\ u, \\
& \quad 1 + (bheight\ t \uparrow bheight\ u)) \\
= & \quad \{\ 將\ balanced\ 與\ bheight\ 的呼叫取出\ \} \\
& \mathbf{let}\ (bt, ht) = (balanced\ t, bheight\ t) \\
& \quad\ (bu, hu) = (balanced\ u, bheight\ u) \\
& \mathbf{in}\ \ (ht == hu \wedge bt \wedge bu, 1 + (ht \uparrow hu)) \\
= & \quad \{\ balHeight\ 之定義\ \} \\
& \mathbf{let}\ (bt, ht) = balHeight\ t \\
& \quad\ (bu, hu) = balHeight\ u \\
& \mathbf{in}\ \ (ht == hu \wedge bt \wedge bu, 1 + (ht \uparrow hu))\ .
\end{aligned}
$$

當輸入為 $\mathsf{R}\ t\ x\ u$ 的情況亦雷同。我們可推導出：

$$
\begin{aligned}
& balHeight :: \mathsf{RBTree} \to (\mathsf{Bool}, \mathbb{N}) \\
& balHeight\ \mathsf{E} = (\mathsf{True}, 0) \\
& balHeight\ (\mathsf{R}\ t\ x\ u) = \mathbf{let}\ (bt, ht) = balHeight\ t \\
& \qquad\qquad\qquad\qquad\quad\ (bu, hu) = balHeight\ u \\
& \qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (ht == hu \wedge bt \wedge bu, (ht \uparrow hu)) \\
& balHeight\ (\mathsf{B}\ t\ x\ u) = \mathbf{let}\ (bt, ht) = balHeight\ t \\
& \qquad\qquad\qquad\qquad\quad\ (bu, hu) = balHeight\ u \\
& \qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (ht == hu \wedge bt \wedge bu, 1 + (ht \uparrow hu))\ .
\end{aligned}
$$

Answer (習題 5.16) — 由 (5.4) 我們推測：

$$
\begin{aligned}
& mdm :: \mathsf{List\ Int} \to (\mathsf{Int} \times \mathsf{Int}) \\
& mdm\ xs = (maxdiff\ xs, minimum\ xs)\ .
\end{aligned}
$$

我們試圖找出 mdm 的歸納定義。顯然 $mdm\ [\,] = (-\infty, \infty)$. 考慮歸納情況 (令 $minus\ (x, y) = x - y$)：

$$
\begin{aligned}
& mdm\ (x : xs) \\
= & \quad \{\ mdm, maxdiff, 與\ minimum\ 之定義\ \} \\
& (maximum\ (map\ minus\ (allpairs\ (x : xs))), x \downarrow minimum\ xs) \\
= & (maximum\ (map\ minus\ (map\ (\lambda y \to (x, y))\ xs +\!\!+ allpairs\ xs)), \\
& \quad x \downarrow minimum\ xs)\ .
\end{aligned}
$$

集中焦點在序對的第一個元素：

$$
\begin{aligned}
& maximum\ (map\ minus\ (map\ (\lambda y \to (x,y))\ xs +\!\!+ allpairs\ xs)) \\
= \quad & \{\ 因\ map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ ys\ (習題\ 2.11)\ \} \\
& maximum\ (map\ minus\ (map\ (\lambda y \to (x,y))\ xs) +\!\!+ \\
& \qquad\qquad map\ minus\ (allpairs\ xs)) \\
= \quad & \{\ map\text{-fusion}\ \} \\
& maximum\ (map\ (x-)\ xs +\!\!+ map\ minus\ (allpairs\ xs)) \\
= \quad & \{\ 因\ maximum\ (xs +\!\!+ ys) = maximum\ xs \uparrow maximum\ ys\ \} \\
& maximum\ (map\ (x-)\ xs) \uparrow maximum\ (map\ minus\ (allpairs\ xs)) \\
= \quad & \{\ 因\ (5.4)\ \} \\
& (x - minimum\ xs) \uparrow maximum\ (map\ minus\ (allpairs\ xs)) \\
= \quad & \{\ maxdiff\ 之定義\ \} \\
& (x - minimum\ xs) \uparrow maxdiff\ xs \quad .
\end{aligned}
$$

回到推導主體：

$$
\begin{aligned}
& mdm\ (x:xs) \\
= \quad & \{\ 上述計算\ \} \\
& ((x - minimum\ xs) \uparrow maxdiff\ xs, x \downarrow minimum\ xs) \\
= \quad & \{\ 使用\ \mathbf{let}\ \} \\
& \mathbf{let}\ (y,z) = (maxdiff\ xs, minimum\ xs) \\
& \mathbf{in}\ ((x-z) \uparrow y, x \downarrow z) \\
= \quad & \{\ mdm\ 之定義\ \} \\
& \mathbf{let}\ (y,z) = mdm\ xs \\
& \mathbf{in}\ ((x-z) \uparrow y, x \downarrow z) \quad .
\end{aligned}
$$

於是我們得知：

$$
\begin{aligned}
mdm\ [\,] \quad & = (-\infty, \infty) \\
mdm\ (x:xs) & = \mathbf{let}\ (y,z) = mdm\ xs \\
& \qquad \mathbf{in}\ ((x-z) \uparrow y, x \downarrow z) \quad .
\end{aligned}
$$

Answer (習題 5.17) — 僅看 $t := \mathsf{Node}\ x\ t\ u$ 的狀況：

$$
\begin{aligned}
& tagsAcc\ (\mathsf{Node}\ x\ t\ u)\ ys \\
= \ & tags\ (\mathsf{Node}\ x\ t\ u) +\!\!+ ys \\
= \ & (tags\ t +\!\!+ [x] +\!\!+ tags\ u) +\!\!+ ys \\
= \quad & \{\ (+\!\!+)\ 的結合律\ \} \\
& tags\ t +\!\!+ (x : tags\ u +\!\!+ ys) \\
= \ & tagsAcc\ t\ (x : tagsAcc\ u\ ys) \quad .
\end{aligned}
$$

我們可導出：

$$
\begin{aligned}
& tagsAcc\ \mathsf{Null} \qquad\quad ys = ys \\
& tagsAcc\ (\mathsf{Node}\ x\ t\ u)\ ys = tagsAcc\ t\ (x : tagsAcc\ u\ ys) \quad .
\end{aligned}
$$

Answer (習題 5.19) — 定義 $tipsAcc\ t\ ys = tips\ t +\!\!+ ys$. 如果 $tipsAcc$ 有個有效率的定義，我們可改定義 $tips\ t = tipsAcc\ t\ [\,]$. 當 $t := \mathsf{Tip}\ x$, $tipsAcc\ (\mathsf{Tip}\ x)\ ys$ 可直接展開為 $x : ys$. 當 $t := \mathsf{Bin}\ t\ u$,

$$
\begin{aligned}
& tipsAcc\ (\mathsf{Bin}\ t\ u)\ ys \\
= & \ (tips\ t +\!\!+ tips\ u) +\!\!+ ys \\
= & \quad \{\ (+\!\!+)\ \text{之結合律}\ \} \\
& \ tips\ t +\!\!+ (tips\ u +\!\!+ ys) \\
= & \ tipsAcc\ t\ (tipsAcc\ u\ ys)\ .
\end{aligned}
$$

因此，

$$
\begin{aligned}
& tipsAcc\ (\mathsf{Tip}\ x)\ \ ys = x : ys \\
& tipsAcc\ (\mathsf{Bin}\ t\ u)\ ys = tipsAcc\ t\ (tipsAcc\ u\ ys)\ .
\end{aligned}
$$

Answer (習題 5.20) — 假設我們要計算 B^{N}:

$$
\begin{aligned}
& b, n, x := \mathsf{B}, \mathsf{N}, 1; \\
& \mathbf{do}\ n \neq 0 \to \mathbf{if}\ even\ n \to b, n := b \times b, n\ `div`\ 2 \\
& \qquad\qquad \mid odd\ \ n \to n, x := n - 1, x \times b \\
& \qquad\qquad \mathbf{fi} \\
& \mathbf{od}; \\
& return\ x
\end{aligned}
$$

其迴圈恆式為 $\mathsf{B}^{\mathsf{N}} = x \times b^n$.

Answer (習題 5.22) — 我們預期將利用 $(\times)$ 的結合律。定義 $factAcc\ n\ y = y \times fact\ n$。顯然 $factAcc\ \mathbf{0}\ y = y$. 至於 $n := \mathbf{1}_+\ n$ 的狀況：

$$
\begin{aligned}
& factAcc\ (\mathbf{1}_+\ n)\ y \\
= & \ y \times fact\ (\mathbf{1}_+\ n) \\
= & \ y \times ((\mathbf{1}_+\ n) \times fact\ n) \\
= & \quad \{\ (\times)\ \text{之結合律}\ \} \\
& \ (y \times (\mathbf{1}_+\ n)) \times fact\ n \\
= & \ factAcc\ n\ (y \times (\mathbf{1}_+\ n))\ .
\end{aligned}
$$

因此我們有了下述定義。由於 $fact\ n = factAcc\ n\ 1$. 這相當於右邊的迴圈：

$$
\begin{aligned}
& factAcc\ \mathbf{0}\ \ \ \ y = y \\
& factAcc\ (\mathbf{1}_+\ n)\ y = \\
& \quad factAcc\ n\ (y \times (\mathbf{1}_+\ n))\ .
\end{aligned}
\qquad\qquad
\begin{aligned}
& n, y := \mathsf{N}, 0; \\
& \mathbf{do}\ n \neq 0 \to n, y := n - 1, y \times n\ \mathbf{od}; \\
& return\ y\ .
\end{aligned}
$$

在推導 $factAcc$ 的歸納定義時，我們不僅用到 $(\times)$ 的結合律，也用到了 $y \times 1 = y$.

Answer (習題 5.23) — 針對 m 做分析。基底情況中，$mulAcc\ 0\ n\ k = k + 0 \times n = k$. 當 m 為非零的偶數，可被改寫為 $2 \times m$ 並演算如下：

$$
\begin{aligned}
& mulAcc\ (2 \times m)\ n\ k \\
= & \ k + (2 \times m) \times n \\
= & \quad \{\ (\times)\ 之交換律與結合律\ \} \\
& \ k + m \times (2 \times n) \\
= & \ mulAcc\ m\ (2 \times n)\ k \quad .
\end{aligned}
$$

當 m 為奇數，可被改寫為 $1 + m$, 並演算如下：

$$
\begin{aligned}
& mulAcc\ (1 + m)\ n\ k \\
= & \ k + (1 + m) \times n \\
= & \quad \{\ (\times)\ 分配進入\ (+)\ \} \\
& \ k + n + m \times n \\
= & \quad \{\ (+)\ 之結合律，mulAcc\ 之定義\ \} \\
& \ mulAcc\ m\ n\ (k + n) \quad .
\end{aligned}
$$

因此我們得到：

$$
\begin{aligned}
& mulAcc\ 0\ n\ k \ = k \\
& mulAcc\ m\ n\ k \mid even\ n = mulAcc\ (m\ `div`\ 2)\ (2 \times n)\ k \\
& \qquad\qquad\quad \mid odd\ n \ = mulAcc\ (m - 1)\ n\ (k + n) \quad .
\end{aligned}
$$

如果改寫成計算 $\mathsf{M} \times \mathsf{N}$ 的迴圈，其恆式為 $\mathsf{M} \times \mathsf{N} = k + m \times n$.

Answer (習題 5.24) — 要使用組對避免重複計算，我們可以定義一個函數同時傳回 $dtoN\ ds$ 以及 10 的 $length\ ds$ 次方。

$$
\begin{aligned}
& dtoNT :: \mathsf{List}\ \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \\
& dtoNT\ ds = (dtoN\ ds, 10^{length\ ds}) \quad .
\end{aligned}
$$

我們可輕易導出：

$$
\begin{aligned}
& dtoNT\ [\,] \quad\ = (0, 1) \\
& dtoNT\ (d : ds) = (d \times t + n, 10 \times t) \quad , \\
& \quad \textbf{where}\ (n, t) = dtoNT\ ds \quad .
\end{aligned}
$$

若使用累積參數，可定義

$$
dtoNAcc\ ds\ n = n \times 10^{length\ ds} + dtoN\ ds \quad .
$$

若有歸納定義，我們可令 $dtoN\ ds = dtoNAcc\ ds\ 0$. 基底狀況為 $dtoNAcc\ [\,]\ n = n$. 至於 $ds := d : ds$ 的情況可演算如下：

$$
\begin{aligned}
& dtoNAcc\ (d : ds)\ n \\
= & \ n \times 10^{length\ (d:ds)} + dtoN\ (d : ds) \\
= & \ n \times 10^{length\ (d:ds)} + d \times 10^{length\ ds} + dtoN\ ds
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\ \mathit{length}\ \text{與乘冪之定義}\ \} \\
&\quad n \times 10 \times 10^{\mathit{length}\ ds} + d \times 10^{\mathit{length}\ ds} + \mathit{dtoN}\ ds \\
&= \quad \{\ \text{算數運算}\ \} \\
&\quad (n \times 10 + d) \times 10^{\mathit{length}\ ds} + \mathit{dtoN}\ ds \\
&= \quad \{\ \mathit{dtoNAcc}\ \text{之定義}\ \} \\
&\quad \mathit{dtoNAcc}\ ds\ (n \times 10 + d)\ .
\end{aligned}
$$

可推導出：

$$
\begin{aligned}
&\mathit{dtoNAcc}\ [\,]\ \quad n = n \\
&\mathit{dtoNAcc}\ (d : ds)\ n = \mathit{dtoNAcc}\ ds\ (n \times 10 + d)\ .
\end{aligned}
$$

Answer (習題 5.25) — 由於我們展開的式子都有 $a \times \mathit{fusc}\ n + b \times \mathit{fusc}\ (1+n)$ 的形式，定義

$$\mathit{fuscAcc}\ n\ a\ b = a \times \mathit{fusc}\ n + b \times \mathit{fusc}\ (1+n)\ .$$

如果 $\mathit{fuscAcc}$ 有歸納定義，我們可令 $\mathit{fusc}\ n = \mathit{fuscAcc}\ n\ 1\ 0$.
為推導 $\mathit{fuscAcc}$, 針對 n 做分析。當 $n := 0$, $\mathit{fuscAcc}\ 0\ a\ b = b$. 當 n 為非零偶數，可被改寫為 $2 \times n$:

$$
\begin{aligned}
&\quad \mathit{fuscAcc}\ (2 \times n)\ a\ b \\
&= a \times \mathit{fusc}\ (2 \times n) + b \times \mathit{fusc}\ (1 + 2 \times n) \\
&= \quad \{\ \mathit{fusc}\ \text{之定義}\ \} \\
&\quad a \times \mathit{fusc}\ n + b \times (\mathit{fusc}\ n + \mathit{fusc}\ (1+n)) \\
&= \quad \{\ \text{四則運算}\ \} \\
&\quad (a+b) \times \mathit{fusc}\ n + b \times \mathit{fusc}\ (1+n) \\
&= \mathit{fuscAcc}\ n\ (a+b)\ b\ .
\end{aligned}
$$

當 n 為奇數，也就是可改寫為 $1 + 2 \times n$ 的形式時：

$$
\begin{aligned}
&\quad \mathit{fuscAcc}\ (1 + 2 \times n)\ a\ b \\
&= a \times \mathit{fusc}\ (1 + 2 \times n) + b \times \mathit{fusc}\ (2 + 2 \times n) \\
&= a \times \mathit{fusc}\ (1 + 2 \times n) + b \times \mathit{fusc}\ (2 \times (1+n)) \\
&= \quad \{\ \mathit{fusc}\ \text{之定義}\ \} \\
&\quad a \times (\mathit{fusc}\ n + \mathit{fusc}\ (1+n)) + b \times \mathit{fusc}\ (1+n) \\
&= \quad \{\ \text{四則運算}\ \} \\
&\quad a \times \mathit{fusc}\ n + (a+b) \times \mathit{fusc}\ (1+n) \\
&= \mathit{fuscAcc}\ n\ a\ (a+b)\ .
\end{aligned}
$$

因此我們有了如下的程式：

$$
\begin{aligned}
&\mathit{fuscAcc}\ 0\ a\ b = b \\
&\mathit{fuscAcc}\ n\ a\ b \mid \mathit{even}\ n = \mathit{fuscAcc}\ (n\ \text{`}\mathit{div}\text{`}\ 2)\ (a+b)\ b \\
&\qquad\qquad\quad \mid \mathit{odd}\ \ n = \mathit{fuscAcc}\ (n\ \text{`}\mathit{div}\text{`}\ 2)\ a\ (a+b)\ .
\end{aligned}
$$

Answer (習題 5.26) — 使用如下的規格：

$$
\begin{aligned}
&mapAcc :: (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b \to \mathsf{List}\ b \\
&mapAcc\ f\ xs\ ys = reverse\ ys +\!\!+ map\ f\ xs \quad .
\end{aligned}
$$

因此 $mapAcc\ f\ [\,]\ ys = reverse\ ys$. 而 $xs := x : xs$ 的情況推導如下：

$$
\begin{aligned}
&\quad mapAcc\ f\ xs\ ys \\
&= reverse\ ys +\!\!+ map\ f\ (x : xs) \\
&= reverse\ ys +\!\!+ (f\ x : map\ f\ xs) \\
&= (reverse\ ys +\!\!+ [f\ x]) +\!\!+ map\ f\ xs \\
&= reverse\ (f\ x : ys) +\!\!+ map\ f\ xs \\
&= mapAcc\ f\ xs\ (f\ x : ys) \quad .
\end{aligned}
$$

Answer (習題 5.27) — 假設習題 5.14 中推導出的函數為 $dd :: \mathsf{ETree}\ a \to (\mathsf{List}\ a, \mathbb{N})$. 我們定義：

$$
\begin{aligned}
&ddD :: \mathsf{ETree}\ a \to (\mathsf{DList}\ a, \mathbb{N}) \\
&ddD\ t = \mathbf{let}\ (xs, n) = dd\ t\ \mathbf{in}\ ((\lambda zs \to xs +\!\!+ zs), n) \quad .
\end{aligned}
$$

有了 ddD 後，我們可改定義 $dd\ t = \mathbf{let}\ (f, n) = ddD\ t\ \mathbf{in}\ (f\ [\,], n)$.
試著推導 ddD 的歸納定義，不難得到 $ddD\ (\mathsf{Tip}\ x) = ((x:), 0)$. 考慮 $ddD\ (\mathsf{Bin}\ t\ u)$, 為計算方便將 dd 改寫為 **if-then-else** 之形式：

$$
\begin{aligned}
&\quad ddD\ (\mathsf{Bin}\ t\ u) \\
&= \quad \{\, ddD \text{ 之定義} \,\} \\
&\quad \mathbf{let}\ (xs, n) = dd\ (\mathsf{Bin}\ t\ u)\ \mathbf{in}\ ((xs +\!\!+), n) \\
&= \quad \{\, dd \text{ 之定義} \,\} \\
&\quad \mathbf{let}\ (xs, m) = dd\ t \\
&\qquad (ys, n) = dd\ u \\
&\quad \mathbf{in}\ (((\mathbf{if}\ m < n\ \mathbf{then}\ xs\ \mathbf{else\ if}\ m == n\ \mathbf{then}\ xs +\!\!+ ys\ \mathbf{else}\ ys) +\!\!+), \\
&\qquad 1 + (m \uparrow n)) \\
&= \quad \{\, \text{函數應用分配入 } \mathbf{if} \,\} \\
&\quad \mathbf{let}\ (xs, m) = dd\ t \\
&\qquad (ys, n) = dd\ u \\
&\quad \mathbf{in}\ (\mathbf{if}\ m < n\ \mathbf{then}\ (xs +\!\!+)\ \mathbf{else\ if}\ m == n\ \mathbf{then}\ (xs +\!\!+ ys +\!\!+)\ \mathbf{else}\ (ys +\!\!+), \\
&\qquad 1 + (m \uparrow n)) \\
&= \quad \{\, \text{抽出 } (xs +\!\!+) \text{ 與 } (ys +\!\!+) \,\} \\
&\quad \mathbf{let}\ (f, m) = (\mathbf{let}\ (xs, m) = dd\ t\ \mathbf{in}\ ((xs +\!\!+), m)) \\
&\qquad (g, n) = (\mathbf{let}\ (ys, n) = dd\ u\ \mathbf{in}\ ((ys +\!\!+), n)) \\
&\quad \mathbf{in}\ (\mathbf{if}\ m < n\ \mathbf{then}\ f\ \mathbf{else\ if}\ m == n\ \mathbf{then}\ f \cdot g\ \mathbf{else}\ g, \\
&\qquad 1 + (m \uparrow n)) \\
&= \quad \{\, ddD \text{ 之定義} \,\} \\
&\quad \mathbf{let}\ (f, m) = ddD\ t
\end{aligned}
$$

```
      (g,n) = ddD u
  in (if m < n then f else if m == n then f · g else g,
      1 + (m ↑ n))  .
```

因此我們得到：

```
ddD (Tip x)   = ((x:),0)
ddD (Bin t u) | m < n = (f,1+n)
              | m == n = (f · g,1+n)
              | m > n = (g,1+m)  ,
  where ((f,m),(g,n)) = (ddD t,ddD u)  .
```

Answer (習題 6.1) —

1. $perms = foldr\ (\lambda x\ xss \to concat\ (map\ (fan\ x)\ xss))\ [[]]$
2. $sublists = foldr\ (\lambda x\ xss \to xss +\!\!+ map\ (x:)\ xss)\ [[]]$
3. $splits$ 可定義如下：

```
splits = foldr spl [([],[])]  ,
  where spl x ((xs,ys):zss) =
              ([],x:xs ++ ys):map ((x:) × id) ((xs,ys):zss)  .
```

Answer (習題 6.2) — 由於 xs 出現在遞迴呼叫以外的地方 — $((y:x:xs):)$, 此處的 fan 定義並不是一個 $foldr$. 但由於 $tail\ (head\ (fan\ y\ xs)) = xs$ （例如，$fan\ 5\ [1,2,3] = [[5,1,2,3],[1,5,2,3],[1,2,5,3],[1,2,3,5]]$, 因此 $tail\ (head\ (fan\ 5\ [1,2,3])) = [1,2,3]$），我們可將 fan寫成：

```
fan y = foldr (λx xss → (y:x:tail (head xss)):map (x:) xss) [[y]]  .
```

Answer (習題 6.4) —

```
  sum · map length
=   { sum = foldr (+) 0, foldr-map 融合 }
  foldr ((+) · length) 0
=   { 摺融合 }
  length · foldr (++) []
= length · concat  .
```

其融合條件證明如下：

```
  length (xs ++ ys)
=   { length 與 (++) 的同態性 }
  length xs + length ys
=   { (·) 之定義 }
  ((+) · length) xs (length ys)  .
```

Answer (習題 6.6) — 回顧：$filter\ p = foldr\ (\lambda x\ xs \to \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)\ [\,]$, $concat = foldr\ (+\!\!+)\ [\,]$. 我們演算如下：

$$
\begin{aligned}
& concat \cdot map\ g \cdot filter\ p \\
= \quad & \{\ \text{摺融合，如下述}\ \} \\
& foldr\ (\lambda x\ ys \to filter\ p\ (f\ x) +\!\!+ ys)\ [\,] \\
= \quad & \{\ foldr\text{-}map\ \text{融合，如下述}\ \} \\
& concat \cdot map\ (filter\ p \cdot f)\ \ .
\end{aligned}
$$

摺融合的條件為：

$$
\begin{aligned}
& concat\ (map\ g\ (\textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)) \\
= \quad & \{\ concat \cdot map\ g\ \text{分配進}\ \textbf{if}\ \text{之中}\ \} \\
& \textbf{if}\ p\ x\ \textbf{then}\ g\ x +\!\!+ concat\ (map\ g\ xs)\ \textbf{else}\ concat\ (map\ g\ xs) \\
= \quad & \{\ \text{提出}\ concat\ (map\ g\ xs)\ \} \\
& (\textbf{if}\ p\ x\ \textbf{then}\ g\ x\ \textbf{else}\ [\,]) +\!\!+ concat\ (map\ g\ xs) \\
= \quad & \{\ \text{假設}\ \} \\
& filter\ p\ (f\ x) +\!\!+ concat\ (map\ g\ xs)\ \ .
\end{aligned}
$$

至於 $foldr\text{-}map$ 融合, 只需驗證 $((+\!\!+) \cdot filter\ p \cdot f)\ x\ ys$ 確實等於 $filter\ p\ (f\ x) +\!\!+ ys$.

Answer (習題 6.7) —
我們需要滿足融合條件 $map\ (\geqslant 0)\ (0 : map\ (x+)\ ys) = step\ x\ (map \geqslant 0\ ys)$ 的 $step$. 演算如下：

$$
\begin{aligned}
& map\ (\geqslant 0)\ (0 : map\ (x+)\ ys) \\
= \ & \mathsf{True} : map\ ((\geqslant 0) \cdot (x+))\ ys \\
= \ & step\ x\ (map\ (\geqslant 0)\ ys)\ \ .
\end{aligned}
$$

然而我們無法由 $map\ (\geqslant 0)\ ys$ 算出 $map\ ((\geqslant 0) \cdot (x+))\ ys$.

Answer (習題 6.8) —

$$
\begin{aligned}
& sum \cdot (+\!\!+ ys) \\
= \ & sum \cdot foldr\ (:)\ ys \\
= \quad & \{\ \text{摺融合}\ \} \\
& foldr\ (+)\ (sum\ ys) \\
= \quad & \{\ \text{摺融合}\ \} \\
& (+(sum\ ys)) \cdot foldr\ (+)\ 0 \\
= \ & (+(sum\ ys)) \cdot sum\ \ .
\end{aligned}
$$

其中第一個摺融合的條件為 $sum\ (x : xs) = x + sum\ xs$ — 我們由此發現融合後的步驟函數為 $(+)$. 第二個摺融合的條件證明如下：

$$
\begin{aligned}
& (+(sum\ ys))\ (x + y) \\
= \ & (x + y) + sum\ ys
\end{aligned}
$$

$$
\begin{aligned}
&= x + (y + sum\ ys) \\
&= x + ((+(sum\ ys))\ y)\ \ .
\end{aligned}
$$

Answer (習題 6.9) — 相當於證明 $length \cdot fan\ y = \mathbf{1}_+ \cdot length$. 推論如下：

$$
\begin{aligned}
&\ \ length \cdot fan\ y \\
&= length \cdot foldr\ (\lambda x\ xss \to (y : x : tail\ (head\ xss)) : map\ (x:)\ xss)\ [[y]]\ \ . \\
&= \quad \{\ \text{摺融合定理}\ \} \\
&\ \ foldr\ \mathbf{1}_+\ 1 \\
&= \quad \{\ \text{摺融合定理}\ \} \\
&\ \ \mathbf{1}_+ \cdot foldr\ \mathbf{1}_+\ \mathbf{0} \\
&= \mathbf{1}_+ \cdot length\ \ .
\end{aligned}
$$

其中第一次融合的融合條件可證明如下：

$$
\begin{aligned}
&\ \ length\ ((y : x : tail\ (head\ xss)) : map\ (x:)\ xss) \\
&= \quad \{\ length\ \text{之定義}\ \} \\
&\ \ \mathbf{1}_+\ (length\ (map\ (x:)\ xss)) \\
&= \quad \{\ length \cdot map\ f = length\ \} \\
&\ \ \mathbf{1}_+\ (length\ xss)\ \ .
\end{aligned}
$$

由此發現步驟函數為 $\mathbf{1}_+$. 第二次融合的融合條件則只需展開定義即可證成。

Answer (習題 6.10) — 基底值為 $base = exp\ b\ 0 = 1$. 為找出步驟函數，我們推論：

$$
\begin{aligned}
&\ \ exp\ b\ (\mathbf{if}\ c\ \mathbf{then}\ 1 + 2 \times n\ \mathbf{else}\ 2 \times n) \\
&= \quad \{\ \text{函數分配進}\ \mathbf{if}\ \} \\
&\ \ \mathbf{if}\ c\ \mathbf{then}\ exp\ b\ (1 + 2 \times n)\ \mathbf{else}\ exp\ b\ (2 \times n) \\
&= \quad \{\ \text{因}\ m^{x+y} = m^x \times m^y\ \} \\
&\ \ \mathbf{if}\ c\ \mathbf{then}\ b \times exp\ b\ (2 \times n)\ \mathbf{else}\ exp\ b\ (2 \times n) \\
&= \quad \{\ \text{因}\ m^{2n} = (m^n)^2\text{, 回顧：}\ square\ x = x \times x\ \} \\
&\ \ \mathbf{if}\ c\ \mathbf{then}\ b \times square\ (exp\ b\ n)\ \mathbf{else}\ square\ (exp\ b\ n)\ \ .
\end{aligned}
$$

因此可得

$$
\begin{aligned}
exp\ b \cdot decimal = foldr\ (\lambda d\ x \to\ &\mathbf{if}\ c\ \mathbf{then}\ b \times square\ x \\
&\mathbf{else}\ square\ x)\ 1\ \ .
\end{aligned}
$$

Answer (習題 6.12) — 回顧 $id = foldN\ (\mathbf{1}_+)\ 0$. 欲將 $fib2$ 融入 id, 基底值為 $fib2\ 0 = (fib\ 1, fib\ 0) = (1, 0)$. 為得到步驟函數，我們演算如下：

$$
\begin{aligned}
&\ \ fib2\ (\mathbf{1}_+\ n) \\
&= (fib\ (\mathbf{1}_+\ (\mathbf{1}_+\ n)), fib\ (\mathbf{1}_+\ n))
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{ fib \text{ 之定義} \} \\
&\quad (fib\ (\mathbf{1}_+\ n) + fib\ n, fib\ (\mathbf{1}_+\ n)) \\
&= (\lambda(x,y) \to (x+y,x))\ (fib2\ n) \quad .
\end{aligned}
$$

因此我們得到

$$
fib2 = foldN\ (\lambda(x,y) \to (x+y,x))\ (1,0) \quad .
$$

Answer (習題 6.13) — 即證明 $length \cdot tags = size$. 回顧 $tags = foldIT\ (\lambda x\ xs\ ys \to xs +\!\!+ [x] +\!\!+ ys)\ [\,]$ 使用摺融合，由於 $length\ [\,] = 0$ 以及 $length\ (xs +\!\!+ [x] +\!\!+ ys) = 1 + length\ xs + length\ ys$, 我們得到 $length \cdot tags = foldIT\ (\lambda x\ m\ n \to 1+m+n)\ 0 = size$.

Answer (習題 6.14) — ETree 上的 $foldET$-$mapE$ 融合定理為：

$$
foldET\ f\ k \cdot mapE\ g = foldET\ f\ (k \cdot g) \quad .
$$

由於 $mapE\ g = foldET\ \mathsf{Bin}\ f$, 欲證明上式可用摺融合定理。其融合條件 $foldET\ f\ k\ (\mathsf{Bin}\ t\ u) = f\ (foldET\ f\ k\ t)\ (foldET\ f\ k\ u)$ 恰巧是 $foldET$ 之定義。

Answer (習題 6.15) — 函數 $mapI$ 可定義如下：

$$
mapI\ f = foldIT\ (\lambda x\ t\ u \to \mathsf{Node}\ (f\ x)\ t\ u)\ \mathsf{Null} \quad .
$$

考慮 $foldIT\ f\ e \cdot mapI\ g$ 之融合。其基底值為 $foldIT\ f\ e\ \mathsf{Null} = e$. 步驟函數的推導如下：

$$
\begin{aligned}
&\quad foldIT\ f\ e\ (\mathsf{Node}\ (g\ x)\ t\ u) \\
&= \quad \{ foldIT \text{ 之定義} \} \\
&\quad f\ (g\ x)\ (foldIT\ f\ e\ t)\ (foldIT\ f\ e\ u) \quad .
\end{aligned}
$$

因此 $foldIT\ f\ e \cdot mapI\ g = foldIT\ (\lambda x\ y\ z \to f\ (g\ x)\ y\ z)\ e$.

Answer (習題 6.16) — 即證明 $minE \cdot mapE\ (x+) = (x+) \cdot minE$. 推論如下：

$$
\begin{aligned}
&\quad minE \cdot mapE\ (x+) \\
&= \quad \{ foldET\text{-}mapE \text{ 融合，見習題 6.14} \} \\
&\quad foldET\ (\downarrow)\ (id \cdot (x+)) \\
&= \quad \{ foldET \text{ 融合，如下述} \} \\
&\quad (x+) \cdot minE \quad .
\end{aligned}
$$

融合的基底函數為 $id \cdot (x+) = (x+) \cdot id$, 融合條件則為 $x + (y \downarrow z) = (x+y) \downarrow (x+z)$。

Answer (習題 6.17) — 回想 $tags = foldIT\ (\lambda x\ xs\ ys \to xs +\!\!+ [x] +\!\!+ ys)\ [\,]$. 融合後之基底值為 $(+\!\!+)\ [\,] = id$. 融合後之步驟函數 $step$ 須滿足 $(+\!\!+)\ (xs +\!\!+ [x] +\!\!+ ys) = step\ x\ (xs +\!\!+)\ (ys +\!\!+)$. 但由於左手邊的 $(+\!\!+)$ 還需一個參數才能化簡，我們在左右兩邊各補上一個參數 zs. 演算如下：

$$
\begin{aligned}
& (+\!\!+)\ (xs +\!\!+ [x] +\!\!+ ys)\ zs \\
= {} & (xs +\!\!+ [x] +\!\!+ ys) +\!\!+ zs \\
= {} & \quad \{\ (+\!\!+)\ \text{之遞移律}\ \} \\
& xs +\!\!+ (x : (ys +\!\!+ zs)) \\
= {} & \quad \{\ (\cdot)\ \text{之定義}\ \} \\
& ((xs +\!\!+) \cdot (x:) \cdot (ys +\!\!+))\ zs \quad .
\end{aligned}
$$

因此我們得到 $(+\!\!+) \cdot tags = foldIT\ (\lambda x\, f\ g \to f \cdot (x:) \cdot g)\ id$.

Answer (習題 6.18) — 基底函數為 $filter\ (all\ ascending) \cdot wrap3 = wrap3$. 為求出步驟函數，我們推論：

$$
\begin{aligned}
& filter\ (all\ ascending)\ (concat\ (map\ (extend\ x)\ ysss)) \\
= {} & \quad \{ filter\ p \cdot concat = concat \cdot map\ (filter\ p),\ map\ \text{融合} \} \\
& concat\ (map\ (filter\ (all\ ascending) \cdot extend\ x)\ ysss) \\
= {} & \quad \{\ \text{推導}\ extendAsc,\ \text{如下述}\ \} \\
& concat\ (map\ (extendAsc\ x)\ (filter\ (all\ ascending)\ ysss)) \quad ,
\end{aligned}
$$

欲使得最後一步成立，我們使用習題 6.6中的性質，試圖找到滿足下述條件的 $extendAsc$：

$$
\begin{aligned}
& filter\ (all\ ascending)\ (extend\ x\ yss) = \\
& \quad \textbf{if}\ all\ ascending\ yss\ \textbf{then}\ extendAsc\ x\ yss\ \textbf{else}\ [\,] \quad .
\end{aligned}
$$

我們演算如下：

$$
\begin{aligned}
& filter\ (all\ ascending)\ (extend\ x\ (ys : yss)) \\
= {} & filter\ (all\ ascending)\ [[x] : ys : yss, (x : ys) : yss] \\
= {} & \quad \{ filter\ \text{之定義; 因}\ ascending\ [x] = \mathsf{True}\ \} \\
& \textbf{if}\ all\ ascending\ (ys : yss)\ \textbf{then} \\
& \quad ([x] : ys : yss) : filter\ (all\ ascending)\ [(x : ys) : yss] \\
& \textbf{else}\ filter\ (all\ ascending)\ [(x : ys) : yss] \\
= {} & \quad \{ filter\ \text{與}\ ascending\ \text{之定義；重安排}\ \textbf{if}\ \text{的幾個分支}\ \} \\
& \textbf{if}\ all\ ascending\ (ys : yss)\ \textbf{then} \\
& \quad \textbf{if}\ x \geqslant head\ ys\ \textbf{then}\ [[x] : ys : yss, (x : ys) : yss] \\
& \qquad\qquad \textbf{else}\ [[x] : ys : yss] \\
& \textbf{else}\ [\,] \\
= {} & \quad \{\ \text{抽取出}\ extendAsc\ \text{如下}\ \} \\
& \textbf{if}\ all\ ascending\ (ys : yss)\ \textbf{then}\ extend'\ x\ (ys : yss)\ \textbf{else}\ [\,]
\end{aligned}
$$

其中 $extendAsc$ 的定義如下：

$$
\begin{aligned}
& extendAsc\ x\ (ys : yss) = \textbf{if}\ x \geqslant head\ ys\ \textbf{then}\ [[x] : ys : yss, (x : ys) : yss] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ [[x] : ys : yss] \quad .
\end{aligned}
$$

Answer (習題 8.2) — 考慮 $eval\ (\mathsf{Add}\ e_0\ e_1)$:

$$
\begin{aligned}
& eval\ (\mathsf{Add}\ e_0\ e_1)\ env \\
= \quad & \{\ eval\ \text{之定義}\ \} \\
& (eval\ e_0 \gg\!= \lambda v_0 \to eval\ e_1 \gg\!= \lambda v_1 \to return\ (v_0 + v_1))\ env \\
= \quad & \{\ (\gg\!=)\ \text{之定義}\ \} \\
& (\lambda v_0 \to eval\ e_1 \gg\!= \lambda v_1 \to return\ (v_0 + v_1))\ (eval\ e_0\ env)\ env \\
= \quad & \{\ \text{函數應用}\ \} \\
& (eval\ e_1 \gg\!= \lambda v_1 \to return\ (eval\ e_0\ env + v_1))\ env \\
= \quad & \{\ (\gg\!=)\ \text{之定義, 函數應用}\ \} \\
& return\ (eval\ e_0\ env + eval\ e_1\ env)\ env \\
= \quad & \{\ return\ \text{之定義}\ \} \\
& eval\ e_0\ env + eval\ e_1\ env \quad .
\end{aligned}
$$

考慮 $eval\ (\mathsf{Let}\ x\ e_0\ e_1)\ env$:

$$
\begin{aligned}
& eval\ (\mathsf{Let}\ x\ e_0\ e_1)\ env \\
= \quad & \{\ eval\ \text{之定義}\ \} \\
& (eval\ e_0 \gg\!= \lambda v \to local\ ((x, v) :)\ (eval\ e_1))\ env \\
= \quad & \{\ (\gg\!=)\ \text{之定義, 函數應用}\ \} \\
& local\ ((x, eval\ e_0\ env) :)\ (eval\ e_1)\ env \\
= \quad & \{\ local\ \text{之定義}\ \} \\
& eval\ e_1\ ((x, eval\ e_0\ env) : env) \quad .
\end{aligned}
$$

Index