Haskell 遇见 Evariste

PAULO R. PEREIRA,Checkmarx，葡萄牙JOSÉ N. OLIVEIRA, HASLab/ U.Minho &INESC TEC，葡萄牙

自 20 世纪 50 年代末作为一门新兴的科学知识领域诞生以来，计算机编程已成为许多其他学科所需的一项基本技能。然而，编程并非易事，容易出错，而代码重用是提高效率的关键。因此，软件程序库需要高质量的文档，但实际情况往往并非如此。本文选取 Hackage 仓库中的若干 Haskell 函数作为案例研究，并将其描述与其他语言中的类似函数进行比较，展示了如何遵循一种称为易难划分 的形式化策略，实现清晰的表达和良好的概念设计。这一策略具有普适性且富有成效，即使非正式地使用也不例外。它易于应用于函数式编程，并可用于程序分析和综合。

CCS 概念： • **软件及其工程** → **函数式语言**； •**计算理论** → **程序规约** **s**.

Add补充关键词与短语：函数式编程、软件库、需求分析、形式化方法

ACM 引用格式：
Paulo R. Pereira 和 José N. Oliveira。2024。《Haskell meets Evariste》。1，1（2024年6月），13页。https://doi.org/--.--

arXiv:2602.16809v1 [cs.PL] 18 Feb 2026

1 引言

编程的本质在于将人类意图转化为机器可执行的指令，这项工作要求表达清晰、规格说明明确无歧义。形式化规格说明 [3], 具有数学性质，最能消除歧义；但面对大型问题时，它们会变得过于复杂，令普通程序员难以理解和使用。

正如函数式程序员所熟知的那样，组合性以及复用软件库提供的（较小）组件能够解决这一难题。然而，程序员如何确定自己导入的函数究竟做什么？如果不使用形式化规格说明，他们就只能依赖文字描述，而这些描述的质量可能差异很大。与工程的其他领域一样，确保软件文档的技术准确性和较高的可读性至关重要。但这正逐渐成为一个问题，因为当今学生（未来的专业程序员）在数学以及写作和阅读方面的能力尚未达到要求。可以通过新颖的编程教学策略克服数学恐惧症，例如 Carroll Morgan 的非正式地进行形式化方法[16]。但如果学生最终连书面文本也无法理解，这种方法就会失败。

这个问题的另一面出现在“正向工程”编程中，例如所谓的需求分析 [23]。在这里，思路清晰同样至关重要，而依赖简单却有效的概念似乎是良好设计的关键 [13]。

本文通过弥合形式描述与非形式描述之间的鸿沟，探讨这一问题的两个方面，研究对象是一类程序，其规格说明符合一种特定但相对宽泛的规格说明模式。我们的分析将

作者联系方式 ：Paulo R. Pereira，Checkmarx，布拉加，葡萄牙，prpereira.hs@gmail.com；José N. Oliveira，HASLab/U.Minho 与 INESC TEC ，布拉加，葡萄牙，j no@di.uminho.pt.

允许为个人或课堂使用而免费制作本作品的全部或部分数字版或纸质副本，前提是这些副本不得用于盈利或商业用途，也不得为此分发，并且副本须包含本声明及首页上的完整引文。对于本作品中由作者以外的其他人拥有版权的部分，必须尊重其版权。注明出处后允许摘录。若要以其他方式复制、再版、发布到服务器或重新分发至列表，须事先获得明确许可和/或支付费用。请向 permissions@acm.org 申请许可。
© 2024 版权归所有者/作者所有。出版权已授权给 ACM。提交至 ACM 的稿件

重点关注函数式编程，尤其是 Hackage 仓库以及其他语言的类似代码仓库中提供的一些列表操作函数。

本文结构如下：简要说明研究动机（第 2 节）之后，第 3 节和第 4 节以 *takeWhile* 函数为例，阐述本文的核心思想，该函数体现了一种所谓的易难划分 函数式设计模式。后续各节将这一示例加以推广，并说明如何从形式上为该方法提供支撑。最后，第 8 节和第 10 节简要讨论了为何清晰性和扎实的基础有助于更好地理解软件。

2 研究动机

让我们分析一下，在一些最知名的编程语言中，某些标准且直观的函数是如何编写文档的。我们先以软件库中一个广为人知的函数——*zip* ——的文档为例。以 Swift 为例，其文本规格说明如下：

> 由两个底层序列构造出一个由成对元素组成的序列。 *[6]*

这种描述留下了很大的诠释空间。例如，$zip\ [1,2,3]\ [a,\ b] = [(1,\ b),\ (3,\ a),\ (3,\ b)\ ]$可以是由序列 $[1,2,3]$和 $[a,\ b]$, $[(2,\ a),\ (1,\ a)\ ]$构造的成对元素序列，这种理解同样可以接受。形式上来说，Swift 中的 *zip* 是一个关系，而非函数，因为对于同一个输入，它可以给出多个可接受的输出。F# 文档中对 *zip* 的描述与 Swift 的基本相同：

> *List.zip* 接收两个单元素列表，并生成一个由元组对组成的列表。 [15]

Elixir 的 Enum 库中的“same”函数描述如下：

> 将有限集合中的可枚举对象的对应元素 *zip* 成一个元组列表 *[8]*。

在对“对应”这一形容词进行合理解释的情况下，Elixir 的 *zip* 比 Swift 和 F# 的实现具有更少的解释自由度，因此排除了 $[(1,b),(3,a),(3,b)]$, 等所有元组。不过， $[(1,a)]$、 $[(2,b)]$,，甚至 $[(2,b),(1,a),(2,b)]$ 仍然是可接受的输出。因此，Elixir 文档中给出的 *zip* 也是一种关系。

Haskell 中也存在类似程度的歧义 's*Data.List* 库：

> 接收两个列表，并返回由对应元素组成的列表。如果其中一个输入列表比另一个短，则较长列表中多出的 *s* 元素会被丢弃，即使其中一个列表是无限的。 *[11]*

请注意，不定冠词“a”带有关系式的解读。不过，提及“多余元素”表明人们多少会关注输出中额外的位置信息。

JavaScript 的描述具体得多，这一点从定冠词的使用中可以看出来：

> *zip* 方法会将给定数组中的值与原集合中对应索引处的值合并在一起。 *[7]*

不过，只要一个集合中的索引无法在另一个集合中找到，这种“the”的用法就会产生问题。因此，JavaScript 中*zip* 的含义是一个偏函数。最后，Python 对*zip* 的定义如下：

> 创建一个迭代器，从每个可迭代对象中聚合元素。返回一个由元组组成的迭代器，其中第 i 个元组包含参数序列或可迭代对象中的第 i 个元素。迭代器会在最短的输入可迭代对象耗尽时停止。只有一个可迭代对象参数时，返回由单元素元组组成的迭代器。没有参数时，返回一个空迭代器。 *[9]*

与之前的描述相比，Python 的描述似乎过于具体，几乎用自然语言描述了实际实现。难道就没有一种介于两者之间、对 *zip* 作出明确说明、又不至于如此具体的非正式描述吗？提交至 ACM 的稿件

答案或许就在 Carroll Morgan 的箴言“以非正式的方式进行形式化”之中， [16], 它概括了即使在文本规格说明中也力求精确的要义，强调严谨表达，同时不牺牲可读性。如何才能做到如此简洁?

3 用非正式的方式谈形式化

如果对 *takeWhile* 函数或方法重复上述练习，就会发现其文字规格说明的质量同样参差不齐。特别是，我们会看到，它在 Swift 中的名称是*prefix*，而 Haskell 的 Data.List 程序包中的描述如下:

> 应用于谓词 p 和列表 xs，返回 xs 中满足 p 的元素所构成的最长前缀（可能为空）。 *[11]* (1)

这段摘录中引人注目的是，它使用了概念 列表前缀，并对前缀之间的关系采用了序关系，由此引出了最高级“最长”。注意这段文字表达得多么清楚：(a) $takeWhile\ p\ xs$ 是 xs 的一个前缀；(b) 这样的前缀中的所有x都满足p；(c) $takeWhile\ p\ xs$ 是这样的最长前缀。还要注意，(a) 和 (b) 中的不定冠词到了 (c) 中就有了确定的指代。[1]

让我们深入分析这些子句。首先，令 $x_1 \preccurlyeq x_2$ 表示列表 x_1 是 x_2的前缀这一事实。那么子句 (a) 就变成了

$$takeWhile\ p\ xs \preccurlyeq xs$$

由条款 (b) 很容易得出： $\forall x : x \in takeWhile\ p\ xs : p\ x = True$。在 Haskell 中使用 $all\ p$函数可以大大简化这一表达式:

$$all\ p\ xs = \forall\ x\ :\ x \in xs :\ p\ x = True \tag{2}$$

我们可以将 (a) + (b) 合并如下:

$$ys = takeWhile\ p\ xs \quad \Rightarrow \quad \begin{cases} ys \preccurlyeq xs \\ all\ p\ ys \end{cases} \tag{3}$$

现在看看条款 (c) 规定了什么：在所有满足 (3) 右侧各条款的前缀中，$ys\ ys$ 是$takeWhile\ p\ xs$的前缀，而后者是这类前缀中最长的。因此，将$ys = takeWhile\ p\ xs$ 替换为$ys \preccurlyeq takeWhile\ p\ xs$时，逆向（⇐）蕴含成立；这就是用符号形式化表述的条款 (c)。由此得到 Hackage 仓库中 Data.List 对*takeWhile* 的（非正式）描述所对应的以下形式化规范:

$$\begin{cases} ys \preccurlyeq xs \\ all\ p\ ys \end{cases} \quad \Leftrightarrow\ ys \preccurlyeq takeWhile\ p\ xs \tag{4}$$

由此我们得出结论，(1) 中清晰明了的文字表述是“非正式地走向形式化”的一个好例子 [16]。

事实证明，(1) 和 (4) 符合一种常见的规格模式：编程问题的描述被分为简单部分——上文的条款 (a) + (b)——它提供许多解（一个二元关系），然后从中选出最佳解——即困难部分 [17]。在上面的*takeWhile* 示例中，困难部分由条款 (c) 给出。

将（1）与其他语言的程序库中同一函数的描述加以比较，颇为有趣:

- JavaScript:

[1]当然，人们 ne eds 必须确信此类最长前缀始终存在，但练习的这一部分属于讨论的另一层面，正如我们将 se e 稍后。

takeWhile 方法会返回集合中的项目，直到给定的回调函数返回 *false*。 *[7]*

- Python：只要谓词为真，就从可迭代对象中返回元素的迭代器。 *[10]*
- Swift：返回一个子序列，其中包含谓词返回 *false* 之前的初始元素，并跳过其余元素[2]。 *[6]*
- Elixir：*take_while*(可枚举对象,函数) 当函数返回真值时，从可枚举对象的开头获取元素。 *[8]*
- F#：*[*使用*]* *Seq.takeWhile* 时，需指定一个谓词函数（布尔函数），并从另一个序列创建一个序列，其中包含原序列中谓词为真的元素；但遇到第一个谓词返回 *false* 的元素时即停止。 *[15]*

显然，JavaScript、Python 和 F# 并未确保 $ys \preccurlyeq xs$ 满足（4）；Swift 使用 子序列 关系，这比前缀更宽泛，因此需要加上“包含初始元素”才能达到所需的精确度；最后，Elixir 的描述更偏算法化，但“从开头获取元素”抓住了前缀这一概念。

事实证明，*takeWhile* (1,4) 中的规格模式是伽罗瓦连接的一个实例。伽罗瓦连接是法国数学家 Evariste Galois（1811—1832）提出的一种数学工具。参考文献 [17, 20] 给出了这一模式的更多实例，并进一步发展了相应的理论。

本文更关注向编程社区，尤其是函数式程序员，传达一个更广泛的信息：库文档需要精确，并且这与形式化正确性息息相关。本文将从 *takeWhile* 的例子出发，展示这种模式的应用价值，并在 Haskell 程序库中寻找它的实例。闲话少说，让我们来一场“Haskell 与 Evariste 的相遇”，看看后者能带来什么启发。

4 *takeWhile*（续）

在上一节中，(1)——分别对应（4）——被视为对函数（方法）*takeWhile* 含义的非正式——分别对应正式——描述；程序员可以将其视为接口，据此判断是否在编写的程序中重用该函数。对函数功能的这种理解与其性质有关。事实上，Haskell 程序库中的许多条目都借助性质来更好地补充给出的文字说明，例如*foldr* 和 *foldl* [11]。

我们主张，诸如 (4) 这样的规格说明——及其以精确文字所作的表述 (1)——扮演着泛性质 的角色，其他性质可由此提取。此外，实现也可以从这类泛性质中推导出来，如 [20, 21] 所示。事实上，稍后我们将证明，标准实现

$$
\begin{aligned}
&takeWhile \quad :: (a \to Bool) \to [a] \to [a] \\
&takeWhile\ \_\ [\,] = [\,] \\
&takeWhile\ p\ (x : xs)
\end{aligned}
$$

[2]如前所述，Swift 的 *takeWhile* 函数称为*prefix*。提交至 ACM 的稿件

```
  | p x          = x : takeWhile p xs
  | otherwise =  [ ]
```

见于 Data.List 包中者可由 (4) 推导得出，这意味着实现与形式化规范在形式上一致。换言之，该代码相对于其形式化规范 (4) 是正确的。

在推导之前，我们先猜想一下 *takeWhile* 可能具备哪些性质，从而丰富它在 Data.List 中的描述。例如，连续应用两个 ***takeWhile*** 会怎样？我们会预期 $takeWhile\ p \cdot takeWhile\ q = takeWhile\ (p \wedge q)$ 成立，其中 $(p \wedge q)\ x = (p\ x) \wedge (q\ x)$——请注意 $(\wedge)$ 这里的重载无伤大雅。用逐点 Haskell 表达同样的猜想如下：

$$takeWhile\ p\ (takeWhile\ q\ xs) = takeWhile\ (p \wedge q)\ xs \tag{5}$$

要认可或否定上述假设，可以尝试从 (1) 开始，但嵌套最高级表达式（“最长者中的最长者”，等等）会造成阻碍。从 (4) 开始则能克服这些障碍，但我们还需要了解有关列表前缀的一些额外形式规则。

我们接受[3] 前缀构成一个偏序，也就是说：(a) 任意列表都是自身的前缀，即 $x \preccurlyeq x$ 始终成立；(b) 前缀的前缀仍是前缀（这一概念的传递性）；(c) “互为前缀”意味着相等：$x \preccurlyeq y$ 且 $y \preccurlyeq x$ 蕴含 $x = y$（反对称性）。

现在设想有两个列表 xs 和 ys，它们拥有**完全相同的** 前缀。也就是说，如果某个 zs 是 xs 的前缀，那么它也是 ys 的前缀，反之亦然。直观地说，xs 和 ys 必定是同一个列表，事实也确实如此。这称为**间接相等** 原理，早在迪杰斯特拉的技术笔记中就能找到 [5], ，当然，这一原理在数学中的应用要古老得多。[4] 用形式化的方式表述，这一原理如下：[5]

$$xs = ys \quad \Leftrightarrow \quad \forall\ zs\ ::\ (zs \preccurlyeq xs \Leftrightarrow zs \preccurlyeq ys) \tag{6}$$

正如我们将简要看到的，这就是我们确认（5）所需知道的全部内容。请注意，我们还没有 *takeWhile* 的 Haskell 代码，因此无法尝试在比如说 Agda 或 Coq 中进行归纳形式证明。我们对 *takeWhile* 的全部了解，就是它出现在（4）中某个前缀符号的右侧。因此我们有两个可能的起点：要么是 $zs \preccurlyeq takeWhile\ p\ (takeWhile\ q\ xs)$，要么是 $zs \preccurlyeq takeWhile\ (p \wedge q)\ xs$。让我们选择第一个选项：

$$\begin{array}{ll}
 & zs \preccurlyeq takeWhile\ p\ (takeWhile\ q\ xs) \\
\Leftrightarrow & \quad \{\ \text{(4) concerning the outermost } takeWhile\ \} \\
 & all\ p\ zs \wedge zs \preccurlyeq takeWhile\ q\ xs \\
\Leftrightarrow & \quad \{\ \text{(4) concerning the other } takeWhile\ \} \\
 & all\ p\ zs \wedge (all\ q\ zs \wedge zs \preccurlyeq xs) \\
\Leftrightarrow & \quad \{\ \text{associativity of conjunction}\ \} \\
 & (all\ p\ zs \wedge all\ q\ zs) \wedge zs \preccurlyeq xs
\end{array}$$

[3]这里我们指的是**非正式地接受**，也就是说，本文没有给出形式化证据。关于此类形式化证据，请参阅 [21]。[4]例如，在经典集合论中，如果两个集合 A 和 B 恰好拥有相同的元素，就称它们相同，也就是说，对于任意 a，$a \in A$ 等价于 $a \in B$。[5]注意，这一原理并非列表前缀关系所独有：它适用于**任意偏序**。

$$
\begin{array}{ll}
\Leftrightarrow & \{\ \text{property of universal quantification, cf (2)}\ \} \\
& (all\ (p\ \wedge\ q)\ zs)\ \wedge\ zs\ \preccurlyeq\ xs \\
\Leftrightarrow & \{\ \text{(4) again}\ \} \\
& zs\ \preccurlyeq\ takeWhile\ (p\ \wedge\ q)\ xs
\end{array}
$$

注意到 zs 在上文中是全称量化的，推理起点与终点的等价关系实例化了（6）的右侧子句。取同一子句的左侧部分，我们立即确认了（5）中所猜测的等式。

人们可能希望为 Data.List 中的 $takeWhile$ 条目添加的另一个性质是 $takeWhile[\,] = [\,]$。_对于这一点，(1) 就已经“足够”了，因为空列表的唯一前缀就是它自身。但 (4) 给出了更简单的证据：

$$
\begin{array}{ll}
& ys\ \preccurlyeq\ takeWhile\ \_\ [\,] \\
\Leftrightarrow & \{\ \text{again (4)}\ \} \\
& ys\ \preccurlyeq\ [\,] \wedge all\ \_\ [\,] \\
\Leftrightarrow & \{\ all\ \_\ [\,] = True\ \text{(empty range in universal quantifier)}\ \} \\
& ys\ \preccurlyeq\ [\,] \\
:: & \{\ \text{indirect equality (6)}\ \} \\
& [\,] = takeWhile\ \_\ [\,]
\end{array}
$$

注意，$takeWhile[\_] = [\,]$ 可以看作我们现在希望从 (4) 推导出的 Haskell 代码的基本情况。通过模式匹配，我们只需填写另一种情况中的省略部分，参见：

$$takeWhile\ p\ (x : xs) = \ldots\ldots$$

按照上述相同的步骤，我们必须从 $ys\ \preccurlyeq takeWhile\ p\ (x : xs)$ 开始，看看会发生什么。但我们还需要了解其他信息：关于 $ys\ \preccurlyeq (x : xs)$，我们能说些什么？

注意，$[\,]\ \preccurlyeq xs$ 始终成立，这意味着空列表是前缀序中的最小元素。因此，要么 $ys\ = [\,]$，这样我们就完成了；要么 ys 应以位于最前面的 x 开头，并且其尾部是 xs 的前缀。（关于这些子句的形式化论证，请参见 [20]。）用符号表示为：

$$ys\ \preccurlyeq\ (x : xs)\ \Leftrightarrow\ ys = [\,]\ \vee\ \langle \exists\, zs\ :\ ys = (x : zs)\ :\ zs\ \preccurlyeq\ xs \rangle \tag{7}$$

利用（7），我们按之前的方法继续：

$$
\begin{array}{ll}
& ys\ \preccurlyeq\ takeWhile\ p\ (x : xs) \\
\Leftrightarrow & \{\ \text{again (4)}\ \} \\
& all\ p\ ys\ \wedge\ ys\ \preccurlyeq\ (x : xs) \\
\Leftrightarrow & \{\ \text{(7) above}\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & all\ p\ ys\ \wedge\ (ys = [\,]\ \vee\ \langle \exists\ zs\ :\ ys = (x : zs)\ :\ zs\preccurlyeq xs\rangle) \\
\Leftrightarrow & \quad \{\ \wedge/\vee \text{ distribution}\ \} \\
 & (all\ p\ ys\ \wedge\ ys = [\,])\ \vee\ \langle \exists\ zs\ :\ ys = (x : zs)\ :\ all\ p\ ys\ \wedge\ (zs\preccurlyeq xs)\rangle \\
\Leftrightarrow & \quad \{\ all\ p\ ys \Leftarrow ys = [\,]\ \} \\
 & ys = [\,]\ \vee\ \langle \exists\ zs\ :\ ys = (x : zs)\ :\ all\ p\ ys\ \wedge\ (zs\preccurlyeq xs)\rangle \\
\Leftrightarrow & \quad \{\ all\ p\ (x : xs)\ \Leftrightarrow\ p\ x\ \wedge\ all\ p\ xs\ \} \\
 & ys = [\,]\ \vee\ \langle \exists\ zs\ :\ ys = (x : zs)\ :\ p\ x\ \wedge\ all\ p\ zs\ \wedge\ zs\preccurlyeq xs\rangle \\
\Leftrightarrow & \quad \{\ (4) \text{ once again}\ \} \\
 & ys = [\,]\ \vee\ \langle \exists\ zs\ :\ ys = (x : zs)\ :\ p\ x\ \wedge\ zs\preccurlyeq takeWhile\ p\ xs\rangle
\end{array}
$$

- 情况$p\ x$ 成立时:

$$
\begin{array}{ll}
 & zs = [\,]\ \vee\ \langle \exists\ zs\ :\ zs = (x : zs)\ :\ zs\preccurlyeq takeWhile\ p\ xs\rangle \\
\Leftrightarrow & \quad \{\ (7)\ \} \\
 & zs\preccurlyeq x : takeWhile\ p\ xs \\
:: & \quad \{\ \text{indirect equality (6)}\ \} \\
 & takeWhile\ p\ (x : xs) = x : takeWhile\ p\ xs
\end{array}
$$

- 分支 $\neg\ (p\ x)$:

$$
\begin{array}{ll}
 & zs = [\,] \\
\Leftrightarrow & \quad \{\ \text{by antisymmetry, since } [\,]\preccurlyeq zs \text{ always holds}\ \} \\
 & zs\preccurlyeq [\,] \\
:: & \quad \{\ \text{indirect equality (6)}\ \} \\
 & takeWhile\ p\ (x : t) = [\,]
\end{array}
$$

因此，由（4）可得 Data.List 中给出的 Haskell 代码。请注意，相对于其规格说明，这段代码是“构造即正确”的。

5 推广

前几节关于 *takeWhile* 的案例研究表明，清晰地描述一段代码的含义，能带来显著成效。清晰明确且易于理解的概念 [13] 发挥着重要作用。（在*takeWhile* 案例中，列表前缀这一概念使描述比其他描述更有效、更严谨。）遗憾的是，软件库中的大多数描述都过于含糊，无法取得同样的效果。

笨拙或含糊的文字会导致误解，更不用说不充分或繁复的形式化规格说明了。许多人都有这样的体会：处理这类复杂的规格说明可能得不偿失，因此对它们望而却步。显然，程序文档存在一道鸿沟——有些文档不够严谨，而另一些则严谨到许多人认为过于复杂。这就是为什么

按照 [17] 提出的易难划分 方式表达问题描述很有用，因为这种方式结构简单，且拥有丰富的（通用）理论，可在不同情境中重复使用。

不过 在列表处理领域，我们注意到，Data.List 中的 *take* 也使用前缀的概念来描述：

> *take n* 应用于列表 *xs* 时，会返回 *xs* 中长度为 *n* 的前缀；如果 $n \geqslant \text{length } xs$，则返回 *xs* 本身。 *[11]*

I不过在这种情况下，文字表述不够完善，而且如 [20], 所示，也可以换一种写法 :

> *take n xs* 应返回 *xs* 中长度不超过 *n* 的最长前缀。

根据这个替代文本，可以立即写出相应的形式描述，

$$\underbrace{\text{length } ys \leqslant n \wedge ys \preccurlyeq xs}_{easy} \quad \Leftrightarrow \quad \underbrace{ys \preccurlyeq take\ n\ xs}_{hard}$$

它遵循与（4）相同的易难划分 模式，具有类似的效果，如 [20] 所示。

回到*zip* 函数。正如第 2 节所示，大多数程序库对它的描述都很差；这次我们反其道而行——先写出形式描述，再次借助前缀这一概念：

$$\begin{cases} \text{map } fst\ zs \preccurlyeq xs \\ \text{map } snd\ zs \preccurlyeq ys \end{cases} \quad \Leftrightarrow \quad zs \preccurlyeq zip\ xs\ ys \tag{8}$$

然后用文字表述：

> *zip xs ys* 是可能的最长列表，其中的元素为成对的值；分别将这些对投影为 *fst* 和 *snd* 后，所得列表是 *xs* 和 *ys* 的前缀。

知道

$$unzip\ zs = (\text{map } fst\ zs, \text{map } snd\ zs) \tag{9}$$

可以将（8）表示如下，

$$\textbf{let } (x, y) = unzip\ zs \textbf{ in } \begin{cases} x \preccurlyeq xs \\ y \preccurlyeq ys \end{cases} \quad \Leftrightarrow \quad zs \preccurlyeq zip\ xs\ ys$$

由此可见，*unzip* 是*zip*的一种逆。但它只是右逆，

$$zip\ xs\ ys = zs \textbf{ where } (xs, ys) = unzip\ zs \tag{10}$$

因为每当对长度不同的列表进行 zip 操作时，逆命题都很容易被否定。

没有 Haskell 中 *zip* 的代码，怎么能确定 (10) 成立呢？多亏了 (9)，我们可以在 (10) 中进行相应的代换，得到：

$$zip\ (\text{map } fst\ zs)\ (\text{map } snd\ zs) = zs \tag{11}$$

让我们把它代入 (8)，看看会发生什么：

$$\begin{aligned} & zs \preccurlyeq zip\ (\text{map}\ fst\ zs)\ (\text{map}\ snd\ zs) \\ \Leftrightarrow \quad & \{\ (8)\ \} \\ & \begin{cases} \text{map}\ fst\ zs \preccurlyeq \text{map}\ fst\ zs \\ \text{map}\ snd\ zs \preccurlyeq \text{map}\ snd\ zs \end{cases} \\ \Leftrightarrow \quad & \{\ x \preccurlyeq x \text{ holds for } \textbf{every}\ x\ \} \\ & True \end{aligned}$$

因此我们得到 $zs \preccurlyeq zip\ (\text{map}\ fst\ zs)\ (\text{map}\ snd\ zs)$，这还不是 (11)，但已经很接近了。（关于如何真正得到 (11)，请参见第7节。）

请注意，我们再次在编码函数*zip* 或了解其 Haskell 代码之前，推导出了它的性质。上面的推导是从已知的伽罗瓦连接推断性质的标准方法之一，称为消去 [1]。本质上，我们进行了相应的代换，使 (8) 的左侧显然成立（依据前缀序的自反性），也就是说，我们将其消去了 。之所以称为左消去，是因为它作用于等价关系的左侧。

为了看看对偶的右消去意味着什么，我们只需在 (8) 中令$zs := zip\ xs\ ys$，得到 n:

$$\begin{cases} \text{map}\ fst\ (zip\ xs\ ys) \preccurlyeq xs \\ \text{map}\ snd\ (zip\ xs\ ys) \preccurlyeq ys \end{cases} \Leftrightarrow zip\ xs\ ys \preccurlyeq zip\ xs\ ys$$

现在右侧显然成立，因此可以消去，剩下：

$$\begin{cases} \text{map}\ fst\ (zip\ xs\ ys) \preccurlyeq xs \\ \text{map}\ snd\ (zip\ xs\ ys) \preccurlyeq ys \end{cases}$$

这只是在说，*zip xs ys* 是对描述中简单部分的一种“解”：对它应用*fst* 或*snd* 投影，就会得到被 zip 的列表的前缀。

6 更改序关系

还有几个类似前例的例子取自 Hackage 的 Data.List 包，并在一篇近期的硕士论文 [21], 中进行了详尽推导。该论文试图探究易难划分模式 [17] 的适用范围有多广，并以伽罗瓦连接理论为支撑。（当前论文为了让更广泛的读者易于理解而有意省略的许多技术细节，都可以在那篇论文中找到。）这项工作从三个形式化程度逐渐提高的层面考察了每个案例：文字描述、形式化的逐点描述（如上所示），以及使用关系代数 [1, 2, 17]给出的形式化无点描述。

一个（意料之中）的结论是，列表前缀序仅适用于那些希望保留输入列表开头处连续元素的函数。Data.List 中对*filter*函数的描述如下：

> *Applied to a predicate and a list, returns the list of those elements that satisfy the predicate [11]* (12)

在大多数情况下，不会产生输入列表的前缀。例如，$filter\ even\ [2,4,5] = [2,4]$是 $[2,4,5]$ 的前缀，但$filter\ odd\ [2,4,5] = [5]$不是。

这是否意味着 *filter* 无法纳入迄今为止使用的 易难划分 描述模式？不是，它可以纳入其中。所需的是另一种概念 ——用子列表 代替前缀。例如， $[5]$ 不是前缀

的 [2, 4, 5] 但是 它是其子列表之一。有趣的是，当前缀让位于子列表后，(4) 中的 *takeWhile* 就“变成了”*filter* ，

$$\begin{cases} ys \sqsubseteq xs \\ all\ p\ ys \end{cases} \Leftrightarrow ys \sqsubseteq filter\ p\ xs \tag{13}$$

其中 $y \sqsubseteq x$ 表示“y 是 x 的子列表”。

因此，对 Haskell 的 Data.List 包中的 *filter* ，更好的描述可以直接采用 *takeWhile* (1) 的描述，只需将“前缀”替换为“子列表”：

> 应用于谓词 p 和列表 xs 时，返回 xs 中由满足 p 的元素构成的最长子列表（可能为空）满足 p。

那么，子列表与前缀有何不同？对于所有 xs， $[\,] \sqsubseteq xs$ 也成立，这意味着空列表再次是该序关系中的最小元素。对于 $ys \sqsubseteq (x : xs)$，我们回看 (7)，稍作修改：

$$ys \sqsubseteq (x : xs) \Leftrightarrow ys \sqsubseteq xs \vee \langle \exists\, zs : ys = (x : zs) : zs \sqsubseteq xs \rangle \tag{14}$$

子列表和前缀都是归纳偏序。前缀会保留（当前）列表的表头，直到决定停止——参见 (7) 中的 $ys = [\,]$ ——而子列表则在归纳的每个阶段决定是否舍弃表头。这一区别会影响 *filter* 的实现，使其与 *takeWhile* 有所不同——见 [21]——但对 (5) 等性质则完全没有影响：将 *takeWhile* 替换为 *filter* 时，(5) 仍然成立。

对 *filter* 的正式描述和非正式描述都能进一步提升人们评估其文档的能力。显然，(12) 对子列表允许排列这一点描述得不够准确：短语“这些元素”并不能确保它们按照原始顺序列出。[6]

从 [21], 中选取最后一个例子，我们来看看：

> *dropWhile p xs* 返回 *takeWhile p xs* *[11]* 之后剩余的后缀。

我们发现，这次还需要另一个概念，即列表的后缀。借助 *takeWhile* 虽然巧妙且准确，但必须假设性质 $takeWhile\ p\ xs +\!\!+ dropWhile\ p\ xs = xs$ 成立。要定义不依赖于 *takeWhile* 的 *dropWhile* ，还必须在列表上定义另一种序关系（后缀），记作 ($\preccurlyeq$)：

$$\begin{cases} s \preccurlyeq [\,] \Leftrightarrow s = [\,] \\ s \preccurlyeq (h : t) \Leftrightarrow s = (h : t) \vee s \preccurlyeq t \end{cases} \tag{15}$$

假设 (15) 成立，可以如下描述 *dropWhile* ：

> *dropWhile p* 会生成输入列表中最长的后缀，其表头不满足谓词 p

由此得到

$$headFails\ p\ z \wedge z \preccurlyeq l \Leftrightarrow z \preccurlyeq dropWhile\ p\ l \tag{16}$$

其中

$$\begin{aligned} &headFails && :: (t \to Bool) \to [t] \to Bool \\ &headFails\ p\ [\,] && = True \\ &headFails\ p\ (h : t) && = \neg\ p\ h \end{aligned}$$

[6] Elixir 对“*filter* (可枚举对象，*fun*) 会对 可枚举对象进行筛选，即只返回那些 *fun* 返回真值的元素” [8] 的描述也存在同样的问题。读者不妨查阅其他语言中对 *filter* 的描述并加以核实。提交至 ACM 的稿件

分别根据（16）和（13）计算 *dropWhile* 和 *filter* 的实现，遵循与上述相同的步骤，可在 [21] 中找到。

7 为什么是伽罗瓦连接

概括而言，上文在 Haskell 函数规格说明中展示的易难划分 设计模式 [17]可归纳为以下等价关系

$$\underbrace{f\ y \leqslant x}_{easy} \quad \Leftrightarrow \quad \underbrace{y \preccurlyeq g\ x}_{hard} \tag{17}$$

其中f：$A \to B$和g：$B \to A$是函数，而类型A和B分别配备了两个偏序（$\preccurlyeq$）和（$\leqslant$）。[7] 在数学中，只要（17）成立，就称f和g“伽罗瓦连接”，而等价关系（17）本身则称为伽罗瓦连接，其名称源自埃瓦里斯特·伽罗瓦（1811—1832），这位法国数学家最早对这一概念作出了贡献。例如，在（10）中，$f = \frown zip$，而$g = unzip$；其中 $\frown zip$ 上方的帽子表示zip 未柯里化。在$take$的例子中，则有$f = \overbrace{\ }^{\frown} take$，以及$g\ ys$ = (length ys,ys)。

伴随 f 与 g 可被视为彼此的“不完全逆”，其含义是，“往返”$f(g\ x) \leqslant x$ 与 $y \preccurlyeq g\ (f\ y)$——即 (17) 的两次消去——一般而言并非等式。

基于伽罗瓦连接的推理在计算中已有许多应用，参见例如参考文献 [1, 4, 17, 18, 22] 等。它们可推广为伴随，后者加深了我们对函数式编程关键特性的理解，参见例如 [12, 20]。其主要优势在于丰富的理论，从而带来极大的思维经济性与有效的推理。

仅举一个简短例子来说明这种丰富性，让我们考虑这些等式

$$g\ (f\ (g\ x)) = g\ x \tag{18}$$

$$f\ (g\ (f\ y)) = f\ y \tag{19}$$

可以证明，对于偏序，它们由 (17) 得出。理论的另一部分是：当 f 为单射时，$g\ (f\ y)\ = y$ 成立[19]。

让我们看看 (19) 对 (10) 有何帮助。既然我们有 *unzip* 的定义——它是 (8) 的简单 伴随——就可以证明它是单射 [19]。这样一来，即使不知道zip的定义，也能得到 (11)：$\frown zip\ (unzip\ xs)\ = xs$。

同理，考虑 f 分别为$takeWhile\ p$ (4)、$filter\ p$ (13) 或$dropWhile\ p$ (16) 的情形。在这些情形下，$f\ ys\ = ys$，因此 (18) 可写为$g\ (g\ x)\ = g\ x$。具有这种性质的函数称为幂等函数。由此可知，*takeWhile p*、*filter p* 和 *dropWhile p* 都是幂等函数，也就是说，对它们应用两次（或更多次）与应用一次的效果相同。我们再强调一次，所有这些结论都由底层理论保证，无需 先知道它们的具体实现。

8 非伽罗瓦连接

仅就其本身而言，*“un”* 这一“*unzip*”前缀的词汇含义表明，该函数是zip的某种逆，这一点我们已在上文从形式和非形式两方面加以确认。这可能会让人以为，类似的术语选择也会得出同样的结论，但事实并非如此。下面是两个被广泛用于

[7] 偏序并非 严格必需，因为预序就足够了 [1]。但与易难划分 一同出现的间接相等需要用到偏序

Haskell 中的字符串处理—— *words* / *unwords* 和 *lines* / *unlines*。非正式地说，这两对函数似乎共享相同的“设计模式”：给定某一类型的对象列表，选取其中一个特定对象作为分隔符，以便将这些列表拆分成列表的列表。

一开始，一切似乎都指向另外两个伽罗瓦连接。然而，仔细检查后会发现，尽管它们最初看起来像是伽罗瓦连接，但实际上两者都不是。

验证这一点的一个简单方法是回到 (18,19)，并以 *words*/*unwords* 这一对为例，其中空格字符充当分隔符。很容易找到反例，使得 $unwords \cdot words \cdot unwords = unwords$ 不成立，而根据 (19) 它本应成立。例如，*unwords* ["Hello "] 会得到 "Hello "，将其传给 *words* 后得到 ["Hello"]，而第二个 *unwords* 最终会将其转换为 "Hello"。如果对于 *words*/*unwords* 这一对来说 (19) 不成立——正如 Haskell 当前的实现那样——那么它们就不能构成伽罗瓦连接，也就是说，它们不能成为 *easy-hard-split* 设计模式的实例。对于 *lines* / *unlines* 这一对，也可以很容易地找到类似的反例。

9 推广

在任何工程领域，根据给定领域中的某些约束找到恰当的近似解，是优秀工程师的标志。此类优化也是运筹学的核心。训练深度神经网络也可视为一种近似过程：通过往返遍历各层来尽量减少分类错误。

通过缩小宽松的规格说明与最佳实现之间的差距（当然也要满足某些约束），来计算出恰当的程序，这确实是诸如 [2]等教科书的精髓，也贯穿于渐进式程序精化理论之中，例如 [14], 。因此，将编程视为实现最佳近似[17] 的艺术，似乎不仅具有教学意义，也符合现实。

值得注意的是，本文提出的方法并不限于列表处理，而列表处理正是上文示例的重点。前缀序和子列表序是树上的通用归纳关系（参见子树等）的实例，可以轻松地用点自由风格定义，例如参见 [19]。

10 总结

软件无疑是现代信息技术社会的关键技术之一。编程越来越多地依赖于巧妙复用 大型软件库和代码仓库中的现有代码。准确了解导入的功能实际执行什么，对软件复用至关重要，因此需要清晰、明确的规格说明。通过形式化方法可以实现严谨的规格说明，但这需要具备逻辑和代数方面的基础。软件文档很少使用形式化规范，而是用散文来描述代码的含义。要让这类文档足够精确，就需要采用一种基于扎实概念 [13]的编程 *go-formal-informally* [16]方法。

本文探讨了一种通用的设计模式，称为易难划分 [17], ，该模式将散文与数学结合起来，用于函数式规范。本文通过 Haskell Data-List 库以及用其他语言编写的类似程序库中的示例，展示了这一模式的作用。这种语言模式说明，代码在执行某项特定任务时如何在某种意义上达到最优。其背后的数学概念是伽罗瓦连接：这是一种丰富的构造，无需显式证明，仅通过分析即可推断出所述代码的性质。

还可以举出更多例子，说明采用这种方法可以省去多少（繁重的）证明工作。将近两个世纪以前，埃瓦里斯特·伽罗瓦本人就对此深感忧虑；他对数学中低效的推理方式颇为批判：

> “有些人 [故意]表面上避免进行任何计算，把用代数可以极简表达的内容改用冗长的句子来描述，从而在运算步骤的长度之外，又添上了不适合表达这些内容的语言长度。这些人落后了一百年。”（*1831*）

致谢

作者感谢Haskell研讨会’24的审稿人就改进本稿所提出的建议。第二作者的工作由通过FCT - 科学技术基金会, I.P.（葡萄牙科学技术基金会）拨付的国家基金资助，属于 IBEX 项目，项目编号为 PTDC/CCI-COM/4280/2021。

参考文献

[1] R.C. Backhouse. 2004. 程序构造数学. 诺丁汉大学. 未出版书稿，608页。可从http: //www.cs.nott.ac.uk/~rcb获取。[2] R. Bird与O. de Moor. 1997. 编程代数. Prentice-Hall.[3] E.M. Clarke与J.M. Wing. 1996. 形式化方法：研究现状与未来方向. *ACM*计算调查 28, 4 (1996), 626–643. doi:10.1145/ 242223.242257[4] P. Cousot与R. Cousot. 1977. 抽象解释：通过构造或逼近不动点对程序进行静态分析的统一格模型. 载于*POPL*. 238–252.[5] E.W. 迪杰斯特拉. 2001. 充实的间接相等. 技术札记EWD 1315-0.[6] Apple开发者文档. 2024. Swift函数zip(_:_:)与前缀(while:)). https://developer.apple.com/documentation/swift 访问日期：2026年2月20日。[7] D. Eckermann. 2024. Collect.js（函数takewhile与zip）. https://collect.js.org/ 访问日期：2026年2月20日。[8] HexDocs Elixir. 2004. Enum. https://hexdocs.pm/elixir/1.12/Enum.html 访问日期：2026年2月20日。[9] Python软件基金会. 2024. Python标准库——内置函数. https://docs.python.org/3.3/library/functions.html 访问日期：2026年2月20日。[10] Python软件基金会. 2024. Python标准库——itertools. https://docs.python.org/3/library/itertools.html 访问日期：2026年2月20日。[11] Haskell软件包仓库（Hackage）. 2001. Data.List（列表上的操作）. https://hackage.haskell.org/package/base-4.19.1.0/docs/Data- List.html 访问日期：2026年2月20日。[12] R. Hinze. 2013. 伴随折叠与展开——一项扩展研究. 计算机程序设计科学 78 (11 2013), 2108–2159. doi:10.1016/j.scico.2012. 07.011[13] D. Jackson. 2021. 软件的本质：概念为何对优秀设计至关重要. 普林斯顿大学出版社.[14] D. Méry. 2009. 面向算法系统的基于精化的指南. 国际软件与信息学期刊 3, 2-3 (2009), 197–239. 6月/9月.[15] Microsoft. 2024. F#语言参考（列表与序列）. https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference 访问日期：2026年2月20日。[16] C. Morgan. 2024. 形式化方法，非正式地：如何编写能够正确运行的程序. 剑桥大学出版社，预计2024年.[17] S.-C. Mu与J.N. Oliveira. 2012. 基于伽罗瓦连接的编程. *JLAP* 81, 6 (2012), 680–704.[18] F. Nielson、H.R. Nielson与C. Hankin. 1999. 程序分析原理. 施普林格，柏林、海德堡. doi:10.1007/978-3-662-03811-6[19] J.N. Oliveira. 2023. 演算式程序设计. 编写中的教材草稿，当前版本：2023年9月. 米尼奥大学（PDF）.[20] J.N. Oliveira. 2023. 伴随为何重要——函数式程序员的视角. 载于*WADT’22 (LNCS)*，A. Madeira与M.A. Martins（编）. 25–59.[21] P.R. Pereira. 2023. 回归基于伽罗瓦连接的编程. 硕士学位论文. 米尼奥大学. https://repositorium.uminho.pt/entities/publication/4529acc6-ac25-4f29-b083-5fb9d98c6d80[22] B. von Karger. 1998. 时序代数. 数学结构与计算机科学 8, 3 (1998), 277–320.[23] K. Wiegers与J. Beatty. 2013. 软件需求，第*3*版. 微软出版社.