

Type-Driven Development with Idris

Edwin Brady



MANNING

Idris 读取-评估-打印循环 (REPL) 提供了几个命令。

下面列出的最常见的命令将在本书的整个过程中进行介绍。

命令	论据	描述
<表达式>	没有任何	显示计算表达式的结果。它包含最近评估的结果的变量。
:t	<表达式>	显示表达式的类型。
:全部的	<名称>	显示具有给定名称的函数是否为总计。
:doc	<名称>	显示名称的文档。
:让	<定义>	添加新定义。
:执行	<表达式>	编译并执行表达式。如果没有给出，则编译并执行 main。
:c	<输出文件>	使用入口点 main 编译为可执行文件。
:r	没有任何	重新加载当前模块。
:l	<文件名>	加载一个新文件。
:模块	<模块名称>	导入一个额外的模块以在 REPL 中使用。
:printdef	<名称>	显示名称的定义。
:apropos	<单词>	搜索给定单词的函数名称、类型和文档。
:搜索	<类型>	搜索具有给定类型的函数。
:浏览	<命名空间>	显示给定中定义的名称和类型命名空间。
:q	没有任何	退出 REPL。

使用 Idris 进行类型驱动开发

使用 Idris 进行类型驱动开发

埃德温·布雷迪



曼宁
庇护岛

有关本手册和其他 Manning 书籍的在线信息和订购,请访问
www.manning.com。如果订购数量多,出版商会为这本书提供折扣。
获取更多资讯,请联系

特卖部
曼宁出版公司
20鲍德温路
邮政信箱 761
纽约州庇护岛 11964
邮箱:orders@manning.com

©2017 Manning Publications Co. 保留所有权利。

未经出版商事先书面许可,不得以任何形式或通过电子、机械、影印或其他方式复制、存储在检索系统中或传输本出版物的任何部分。

制造商和销售商用来区分其产品的许多名称都被称为商标。如果这些名称出现在书中,并且 Manning Publications 知道商标声明,则这些名称以首字母大写或全部大写形式印刷。

◎ 认识到保存已写内容的重要性,Manning 的政策是将我们出版的书籍印刷在无酸纸上,为此我们尽最大努力。

还认识到我们有责任保护地球资源,曼宁书籍
印刷在至少 15% 回收和加工且不使用元素氯的纸张上。



曼宁出版公司
20 Baldwin Road
PO Box 761 Shelter
Island, NY 11964

开发编辑:Dan Maharry
评论编辑:Aleksandar Dragosavljevic
技术开发编辑:Andrew Gibson
项目编辑:凯文沙利文
文案编辑:安迪卡罗尔
校对:凯蒂·坦南特
技术校对员:Arnaud Baily,Nicolas Biri
排字员:多蒂·马西科
封面设计师:玛丽亚·都铎

国际标准书号 9781617293023
在美利坚合众国印刷
1 2 3 4 5 6 7 8 9 10 – EBM – 22 21 20 19 18 17

简要内容

第1部分引言	1
1 ■概述 3	
2 ■Idris 25 入门	
第2部分核心IDRIS	53
3 ■使用类型的交互式开发 55	4 ■用户定义的数据类型 87
5 ■交互式程序:输入和输出处理	
123 6 ■使用一等类型编程 147	7 ■接口:使用受约束的泛型类型 182
8 ■等式:表达之间的关系数据 208	9 ■谓词:以类型表达假设和契约 236
10 ■视图:扩展模式匹配 258	
第3部分IDRIS 和现实世界	289
11 ■流和进程:处理无限数据 291	12 ■使用状态编写程序 324
13 ■状态机:验证类型中的协议 352	14 ■相关状态机:处理反馈和错误 373
15 ■类型安全的并发编程 403	

内容

前言 xv 致谢
xvii 关于本书 xix

关于作者二十三
关于封面插图 xxiv

第1 部分 引言 1

1 概述 3

1.1 什么是类型? 4 1.2

引入类型驱动开发 5

矩阵运算 6 ■自动柜员机 7

并发编程 9 ■类型、定义、提炼:类型驱动开发的过程

10 ■依赖类型 11

1.3 纯函数式编程 13 纯度和引用透明性 13 ■

副作用程序 14 ■部分和全部功能 16 1.4 Idris 快速

浏览 17 交互环境 17 ■检查类型 18 编译和运行 Idris

程序 19 ■不完整的定义:使用孔 20 ■一流的类型 22

1.5 总结 24

2伊德里斯 25 入门

2.1 基本类型 26数值类

型和值 27 ■使用强制类型转换的类型转换 28 ■字符和字符串 29 ■布尔值 30 2.2 函数 :ldris 程序的构建块 30

函数类型和定义 31 ■部分应用函数 33 ■编写通用函数 :在 types 33 编写具有约束类型的泛型函数 35 高阶函数类型 36 ■匿名函数 38 局部定义 :let 和 where 39

2.3 复合类型 40元组 40 ■列表

41 ■带有列表的函数 43

2.4 一个完整的 Idris 程序 46空白的意义 :布

局规则 46 ■文档注释 47 ■交互式程序 48

2.5 总结 52

第2 部分核心IDRIS 53

3与类型 55 的交互开发

3.1 Atom 中的交互式编辑 56交互式命令摘

要 57 ■通过模式匹配定义函数 57 ■数据类型和模式 61

3.2 增加类型的精度 :使用向量 64细化 allLengths 的类型 65 ■类型

导向搜索 :自动细化 69 ■类型、定义、细化 :对向量进行排序

70 3.3 示例 :矩阵函数的类型驱动开发 75矩阵运算和它们的类型

76 ■转置矩阵 77 3.4 隐式参数 :类型级变量 82

需要隐式参数 82 ■绑定和未绑定的隐式 83 ■在函数中使用
隐式参数 84 3.5 总结 86

4用户定义的数据类型 87

4.1 定义数据类型 88枚举 89 ■联

合类型 90 ■递归类型 92 通用数据类型 95

4.2 定义依赖数据类型 102

第一个例子:按电源分类车辆 102
定义向量 104 ■使用有界数索引向量 107

4.3 交互式数据存储的类型驱动实现 110

表示商店 112 ■交互式维护 main 中的状态 113 ■命令:解
析用户输入 115
处理命令 118

4.4 总结 122

5 交互程序:输入输出处理 123

5.1 使用 IO 124 进行交互式编程

评估和执行交互式程序 125
动作和排序:>>= 运算符 127
使用 do 表示法 129 进行排序的语法糖

5.2 交互程序和控制流程 132

在交互式定义中产生纯值 132
模式匹配绑定 134 ■使用循环编写交互式定义 136

5.3 读取和验证依赖类型 138

从控制台读取 Vect 139 ■读取未知长度的 Vect 140
■依赖对 141
验证 Vect 长度 143

5.4 总结 146

6 用一流的类型编程 147

6.1 类型级函数:计算类型 148

类型同义词:为复杂类型提供信息性名称 149 ■
具有模式匹配的类型级函数 150 ■在类型中使用
大小写表达式 153

6.2 定义具有可变数量参数的函数 155

加法函数 155 ■格式化输出:类型安全的 printf 函数 157

6.3 使用模式增强交互式数据存储 161

细化 DataStore 类型 162 ■使用 DataStore 的记录 164 ■
使用孔更正编译错误 165
显示存储中的条目 170 ■根据

架构 171 ■更新架构 175 ■使用可能使用 do 表示法对表达式进行排序 177

6.4 总结 181

7 接口 : 使用受约束的泛型类型 182

7.1 与 Eq 和 Ord 183 的一般比较

使用 Eq 测试相等性 183 ■使用接口和实现定义 Eq 约束 185

默认方法定义 189 ■受限实现 189 ■受限接口 : 使用 Ord 定义排序 191

7.2 Prelude 194 中定义的接口

使用 Show 转换为字符串 194 ■定义数字类型 195 ■在类型之间转换使用 Cast 198

7.3 类型参数化的接口 -> 类型 199

使用 Functor 200 在结构中应用函数

使用 Foldable 201 减少结构 ■使用 Monad 和 Applicative 205 的通用 do 表示法

7.4 总结 207

8 平等 : 表达数据之间的关系 208

8.1 保证具有相等类型的数据的等价性 209

实现精确长度,第一次尝试 210 ■将 Nats 的相等性表示为类型 211 ■测试 Nats 的相等性 212 ■作为证明的函数 : 操纵相等性 215

实现 exactLength,第二次尝试 216 ■一般相等 := 类型 218

8.2 实践中的平等 : 类型和推理 220

反转向量 220 ■类型检查和评估 221

重写构造 : 使用相等性重写类型 223

委托证明和重写漏洞 224 ■附加向量,重新访问 225

8.3 空类型和可判定性 227

Void : 没有值的类型 228 ■可判定性 : 以精度检查属性 229 ■

DecEq : 可判定相等的接口 233

8.4 总结 234

9谓词:在类型 236 中表达假设和契约

9.1 隶属度测试:Elem 谓词 237

从 Vect 中删除元素 238 ■ Elem 类型:保证值在向量中 239
 ■从 Vect 中删除元素:类型为契约 241 ■自动隐式参数:自动构造证明
 244 ■可判定谓词:决定 a 矢量 245

9.2 用类型表示程序状态:猜谜游戏 250

表示游戏的状态 250 ■顶级游戏函数 251 ■验证用户输入
 的谓词:ValidInput 251 ■处理猜测 253 ■确定输入有效
 性:检查 ValidInput 255 ■完成顶级游戏实现 255

9.3 总结 257

10次浏览:扩展模式匹配 258

10.1 定义和使用视图 259

匹配列表中的最后一项 260 ■构建视图:覆盖函数 262 ■使
 用块:扩展模式匹配的语法 262 ■示例:使用视图反转列表 264
 ■示例:合并排序 266

10.2 递归视图:终止和效率 271

“Snoc”列表:反向遍历列表 271 ■递归视图和 with 构造 274 ■遍
 历多个参数:嵌套块 275 ■更多遍历:Data.List.Views 277

10.3 数据抽象:使用视图隐藏数据结构 280

题外话:Idris 中的模块 280 ■数据存储,重新访问 282
 ■使用视图遍历存储的内容 284

10.4 总结 288

第3部分IDRIS 和现实世界.....289

11流和进程:处理无限数据 291

11.1 流:生成和处理无限列表 292

标记列表中的元素 293 ■生成无限的数字列表 295 ■题外话:函数

完全吗? 296 ■处理无限列表 297 ■流数据类型
299 ■使用随机数流的算术测验 301

11.2 无限进程:编写交互式总程序 305

描述无限进程 306 ■执行无限进程 307 ■将无限进程
作为总函数执行 308 ■使用惰性类型生成无限结构
309 ■为 InfIO 扩展 do 表示法 311

总算术测验 311

11.3 带终端的交互式程序 314

优化 InfIO:引入终止 314 ■特定于域的命令 317 ■使用
do 表示法对命令进行排序 320

11.4 总结 323

12 编写状态为 324 的程序

12.1 使用可变状态 325

树遍历示例 326 ■使用对表示可变状态 328 ■状态,用
于描述有状态操作的类型 329 ■使用状态进行树遍历 331

12.2 State 333 的自定义实现

定义状态和运行状态 333 ■定义函子,
状态 335 的应用程序和 Monad 实现

12.3 带状态的完整程序:使用记录 340

带状态的交互式程序:重新审视算术测验 340 ■
复杂状态:定义嵌套记录 343
更新记录字段值 344 ■通过应用函数更新记录字段 346
■实施测验 346
运行交互式和有状态程序:执行测验 348

12.4 总结 351

13 状态机:验证类型 352 中的协议

13.1 状态机:跟踪类型 353 中的状态

有限状态机:将门建模为 354 型
门操作序列的交互式开发 356
无限状态:为自动售货机建模 358
经验证的自动售货机描述 360

13.2 状态中的依赖类型 : 实现栈 363

在状态机中表示堆栈操作 364

使用 Vect 366 实现堆栈 ■ 交互使用堆栈 : 基于堆栈的计算器
367

13.3 总结 371

14 相关状态机 : 处理反馈和错误 373

14.1 处理状态转换中的错误 374

改进门模型 : 表示失败 375 ■ 已验证、错误检查、门协议描述 378

14.2 类型中的安全属性 : ATM 382 建模

定义 ATM 383 的状态 ■ 定义 ATM 384 的类型 ■ 在控制台模拟
ATM : 执行 ATMCmd 387 ■ 使用自动隐式优化前提条件 388

14.3 一个经过验证的猜谜游戏 : 用类型 390 描述规则

定义一个抽象的游戏状态和操作 391 ■ 定义游戏状态的类型 392 ■ 实
现游戏 395
定义具体的游戏状态 397 ■ 运行游戏 : 执行 GameLoop 399

14.4 总结 402

15 类型安全的并发编程 403

15.1 Idris 404 中并发编程的原语

定义并发进程 406 ■ 通道库 : 原始消息传递 407 ■ 通道问题 : 类
型错误和阻塞 410

15.2 定义安全消息传递的类型 411

在类型 412 中描述消息传递过程 ■ 使用 Inf 415 使过程总计 ■ 使用
状态机和 Inf 418 保证响应 ■ 通用消息传递过程 422 ■ 为过程定义模
块 426 ■ 示例 1 :

列表处理 427 ■ 示例 2 : 字数统计处理 429

15.3 总结 433

附录 A 安装 Idris 和编辑器模式 435

附录 B 交互式编辑命令 438

附录 C REPL 命令 439

附录 D 延伸阅读 441

索引 445

Machine Translated by Google

前言

计算机无处不在,我们每天都依赖软件。除了运行我们的台式电脑和笔记本电脑外,软件还控制着我们的通信、银行、交通

基础设施,甚至我们的家用电器。即便如此,它仍然被认为是生活中的事实

那个软件不可靠。如果笔记本电脑或手机出现故障,只会带来不便

并且需要重新启动(可能伴随着对失去最后几分钟工作的诅咒)。另一方面,如果控制关键业务应用程序或服务器的软件出现故障,则可能会损失大量时间和金钱。对于安全关键系统,

后果可能更糟。

因此,多年来,计算机科学研究人员一直在寻找

提高软件的健壮性和安全性的方法。许多方法中的一种方法是

使用类型来描述程序应该做什么。特别是,通过使用依赖类型,您可以描述程序的精确属性。这个想法是,如果你能

在其类型中表达程序的意图,并且程序成功进行类型检查,

那么程序必须按预期运行。Idris 编程语言的一个重要(如果是雄心勃勃和长期的)目标是使这项研究的结果

一般软件开发人员都可以访问,并相应地减少关键软件故障的可能性。

最初,本书的重点是在 Idris 中编程:展示如何使用它的类型

系统来保证程序的重要属性。在开发编辑 Dan Maharry 的指导下,在技术开发编辑 Andrew Gibson 的努力下,它已经发展成为与

使用依赖类型进行编程,以了解生成的程序如何工作。你会

了解依赖类型的基础知识,如何使用类型以交互方式定义程序,以及如何根据您对

问题演变。您还将了解类型驱动的一些实际应用
开发,特别是在处理状态、协议和并发方面。

Idris 本身是我自己对依赖类型的程序验证和语言设计进行研究的结果。在花了几个月时间
沉浸在依赖类型编程的概念中之后,我觉得需要一种语言

专为开发人员和研究人员设计。我希望你在学习 Idris 的类型驱动开发时和我在开发它时一样
开心!

致谢

许多人在本书的编写过程中提供了帮助,如果没有他们,这本书就不会存在。特别要感谢 Dan Maharry,他鼓励我更清楚地揭示类型驱动开发的思想。贯穿全书的“类型、定义、改进”的口头禅是丹的建议。我还要非常感谢 Andrew Gibson,他一丝不苟地完成了本书中的所有示例和练习,确保它们有效,检查练习是否可解,并建议对文本和解释进行许多改进。总的来说,我要感谢 Manning Publications 的团队帮助使这本书成为现实。

Idris 的设计很大程度上归功于几十年来对类型理论、函数式编程和语言设计的研究。我特别感谢 James McKinna 和 Conor McBride 在我还是杜伦大学研究生时教我类型理论的基础知识,以及他们从那时起一直提供的建议和鼓励。我还要感谢负责启发我工作的语言和系统的研究人员和开发人员,即 Haskell、Epigram、Agda 和 Coq 等工具。没有以前的工作,伊德里斯就不可能存在,我只能希望它反过来能在未来激励其他人。有关启发 Idris 的工作的一些参考资料,请参见附录 D。

圣安德鲁斯大学和其他地方的几位同事和学生对早期的章节草稿提供了有用的反馈,并且在我写这本书而不是做其他事情时一直很耐心。我要特别感谢 Ozgur Akgun、Nicola Botta、Sam Elliot、Simon Fowler、Nicolas Gaglani (他们为您将在本书中使用的 Atom 编辑器提供了扩展)、Jan de Muijnck-Hughes、Markus Pfeiffer、Chris Schwaab 和 Matú Tejic v ák

为他们的意见和建议。我向其他我忘记十个名字的人表示诚挚的歉意！

购买了早期访问权限的读者和早期草稿的审阅者提供了许多有用的意见和建议。这些评论者包括亚历山大

A. Myltsev, Álvaro Falquina, Arnaud Bailly, Carsten Jørgensen, Christine Koppelt,
乔瓦尼·鲁杰罗、伊恩·迪斯、胡安·加布里埃尔·波诺、马蒂亚斯·伦德尔、菲尔·德茹、
Rintcius Blok, Satadru Roy, Sergey Selyugin, Todd Fine 和 Vitaly Bragilevsky。

我不可能自己实现 Idris。自从我开始开发电流
2011 年末的版本，有很多贡献者，但最重要的是我想
感谢 David Christiansen，他负责 Idris REPL 中的大部分润色工作
和交互式编辑工具；他还努力帮助新人
项目。还要感谢其他贡献者：Ozgur Akgun, Ahmad Salim
Al-Sibahi, Edward Chadwick Amsden, Michael R. Bernstein, Jan Bessai, Nicola Botta,
维塔利·布拉吉列夫斯基、雅各布·布伦克、阿丽莎·卡特、卡特·夏博诺、亚伦·克雷利厄斯、
杰森·达吉特、亚当·桑德伯格·埃里克森、古列尔莫·法奇尼、西蒙·福勒、扎克
邻居肖恩·亨特 Cezar Ionescu Heath Johns 艾琳·克纳普 Paul Koerbitz Niklas
拉尔森, Shea Levy, Mathnerd314, Hannes Mehnert, Mekeor Melire, Melissa Mozifian,
Jan de Muijnck-Hughes, Dominic Mulligan, Echo Nolan, Tom Prince, raichoo, Philip
Rasmussen, Aistis Raulinaitis, Reynir Reynisson, Seo Sanghyeon, 本杰明·桑德斯、
Alexander Shabalin, Jeremy W. Sherman, Timo Petteri Sinnemäki, JP Smith, 令人吃惊，
Chetan T, Matú Teji c v ák, Dirk Ullrich, Leif Warner, Daniel Waterworth, Eric
Weinstein, Jonas Westerlund, Björn Aili, and Zheng Jihui.

最后，感谢我的父母，他们在 1983 年购买了一台 BBC Micro，让我开始了
在这条路上；还有艾玛，她如此耐心地等待我完成这件事，并给我端来咖啡让我继续前进。

关于这本书

使用 Idris 进行类型驱动开发是为了让类型为您工作。类型通常被视为检查错误的工具，程序员首先编写一个完整的程序并使用类型检查器来检测错误。在类型驱动的开发中，你使用类型作为构建程序的工具，使用类型检查器作为你的助手来指导你完成一个完整的工作程序。

本书首先描述了你可以用类型表达什么；然后，它介绍 Idris 编程语言的核心特性。最后，描述了类型驱动开发的一些更实际的应用。

谁应该读这本书

本书面向希望了解使用最新技术的开发人员。复杂的类型系统，以帮助开发强大的软件。它旨在为依赖类型提供易于理解的介绍，并展示现代基于类型的技术如何应用于现实世界的问题。

理想情况下，读者应该已经熟悉函数式编程概念。例如闭包和高阶函数，尽管本书介绍了这些和其他必要的概念。了解另一种函数式编程语言，例如 Haskell、OCaml 或 Scala 将特别有用，尽管没有假设。

路线图

本书分为三个部分。第 1 部分（第 1 章和第 2 章）介绍了概念并介绍了 Idris 编程语言：

第 1 章介绍了类型驱动开发，并让您开始使用伊德里斯环境。

第 2 章涵盖了 Idris 编程的基础知识,包括原始类型和构建 Idris 程序。

第 2 部分 (第 3-10 章)介绍了 Idris 的核心语言特性:

第 3 章讨论使用 Atom 编辑器的交互式开发和

描述了如何使用更精确的类型意味着类型检查器可以提供帮助你写程序。

第 4 章解释了如何定义自己的数据类型,并给出了第一个以类型驱动风格编写大型交互式程序的示例。第 5 章更深入地描述了交互式程序,包括如何使用

帮助验证用户对交互式程序的输入的类型。

第 6 章介绍类型级编程,展示如何编写函数

计算类型以及如何在实践中使用它们。第 7 章描述了如何使用接口来编写具有泛型类型的程序。第 8 章解释了如何使用类型来表达数据之间的关系,

特别是描述数据的属性并保证功能以某种方式行事。

第 9 章进一步解释了类型如何表达函数必须的契约

满足,包括一个展示如何使用类型来描述状态的示例一个系统。

第 10 章介绍了视图,它们是检查和遍历数据结构的替代方法。

第 3 部分 (第 11-15 章)描述了 Idris 在实际软件中的一些应用发展,特别是与国家和互动项目合作:

第 11 章描述了如何处理潜在的无限数据,例如流,以及如何编写和推理可能的交互式程序无限期地运行。

第 12 章解释了如何编写带状态的程序以及如何表示和使用记录操作复杂的状态。

第 13 章展示了如何以 Idris 类型表示状态机,以及如何使用类型来保证程序正确地遵循协议。第 14 章描述了更复杂的状态机,如何处理错误

和来自环境的反馈,以及如何表示安全属性其类型的系统。第

15 章以一个工作示例结束本书:用于并发编程的小型库的类型驱动开发。

一般来说,每一章都建立在前几章介绍的概念之上,所以它是旨在让您按顺序阅读各章。最重要的是,这本书描述了类型驱动开发和交互式构建程序的过程类型。因此,我强烈建议在计算机上完成这些示例

当你阅读。此外,如果您正在阅读电子书,请输入示例 不要只是复制和粘贴。

每章都有练习,因此,在阅读时,请确保完成练习以加强理解。示例解决方案可从本书的网站
[www.manning.com/books/type-driven-development-with-idris 在线获得。](http://www.manning.com/books/type-driven-development-with-idris)

有四个附录:附录 A 描述了如何安装 Idris 和 Atom 编辑器模式,我们将在整本书中使用它们。附录 B 总结了 Atom 支持的交互式编辑命令。附录 C 总结了您可以在 Idris 环境中使用的命令。最后,附录 D 提供了一些启发 Idris 的工作的参考资料,您可以在其中了解有关理论背景和相关工具的更多信息。

代码约定和下载

本书包含许多源代码示例,包括编号列表和内嵌普通文本。在这两种情况下,源代码都被格式化为这样的固定宽度字体,以将其与普通文本分开。

在许多情况下,原始源代码已被重新格式化;我添加了换行符并重新设计了缩进以适应书中可用的页面空间。此外,在文本中描述代码时,源代码中的注释通常已从列表中删除。代码注释伴随着许多清单,突出了重要的概念。

本书中的所有代码都可以从本书的网站(www.manning.com/books/type-driven-development-with-idris)在线获得,并且已经使用 Idris 1.0 进行了测试。

该代码也可在此处的 Git 存储库中找到: <https://github.com/edwinb/TypeDD-Samples>。

在线作者

购买Idris 的 Type-Driven Development 包括免费访问 Manning Publications 运营的私人网络论坛,您可以在该论坛上对本书发表评论、提出技术问题并从作者和其他用户那里获得帮助。要访问论坛并订阅它,请将您的网络浏览器指向www.manning.com/books/type-driven-development-with-idris。此页面提供有关注册后如何进入论坛的信息、可用的帮助类型以及论坛上的行为规则。

Manning 对读者的承诺是提供一个场所,让读者之间以及读者与作者之间进行有意义的对话。

这不是作者对任何特定参与程度的承诺,作者对论坛的贡献仍然是自愿的(并且是无偿的)。我们建议您尝试向他提出具有挑战性的问题,以免他的兴趣偏离!

作者在线论坛和以前讨论的档案将被访问
只要这本书还在印刷,就可以从出版商的网站上下载。

其他在线资源

如果您想了解有关 Idris 的更多信息,可以在 Idris 网站上找到更多资源:<http://idris-lang.org/>。您还可以在其他几个地方找到帮助:

idris-lang Google Group 是一个讨论 Idris 各个方面的活跃小组。

该小组欢迎初学者和更高级的用户提出问题。在 irc.freenode.net 上有一个IRC频道,#idris,同样对问题。

您可以使用 Stack Overflow 上的 Idris 标签提问和回答问题。

关于作者



EDWIN BRADY领导 Idris 编程语言的设计和实施。他是苏格兰圣安德鲁斯大学计算机科学系的讲师,经常在会议上发表演讲。当他不这样做时,你可能会发现他在下围棋,看板球比赛,或者在苏格兰中部的某个小山上。

关于封面插画

Idris 的 Type-Driven Development 封面上的人物标题为“La Gas conne”或“来自加斯科尼的女人”。插图取自多位艺术家的作品集,由 Louis Curmer 编辑并于 1841 年在巴黎出版。该集的标题是 *Les Français peints par eux-mêmes*, 翻译为“自己画的法国人”。每幅插图都是手工精心绘制和着色的, 收藏中丰富多样的图画生动地提醒我们, 仅仅在 200 年前, 世界各地、城镇、村庄和社区的文化差异。人们彼此孤立, 说着不同的方言和语言。无论是在街头还是在乡下, 仅仅通过他们的着装就很容易识别出他们住在哪里, 以及他们的职业或生活站是什么。

从那时起, 着装规范发生了变化, 当时如此丰富的地区多样性已经消失。现在很难区分不同大陆的居民, 更不用说不同的城镇或地区了。也许我们已经用文化多样性换取了更多样化的个人生活 当然是换取了更多样化和快节奏的技术生活。

在很难区分一本计算机书籍与另一本计算机书籍的时代, 曼宁以两个世纪前丰富多样的区域生活为基础的书籍封面庆祝计算机业务的创造性和主动性, 并通过收藏中的图片恢复生机, 例如作为这个。

第1部分

介绍

在 第一部分,您将开始使用 Idris 并了解背后的想法
类型驱动开发。我将带你简要介绍一下 Idris 环境,
你会写一些简单但完整的程序。

在第一章中,我将更多地解释我所说的类型驱动开发的含义。最重要的是,我将定义
我所说的“类型”并给出几个
如何使用表达类型来描述预期目的的示例
你的程序更精确。我还将在 Idris 语言的两个最显着的特征: holes, 它代表尚未完
成的程序的一部分。
被编写,并使用类型作为一流的语言结构。

在深入了解 Idris 中的类型驱动开发之前,牢牢掌握该语言的基础知识是很重要的。
因此,在
第 2 章,我将讨论一些原始语言结构,其中很多
你会熟悉其他语言,并展示如何在 Idris 中构建完整的程序。

Machine Translated by Google

1 概述

本章涵盖

引入类型驱动开发纯函数式编程的精髓
Idris 的第一步

本书介绍了一种使用 Idris 编程语言构建健壮软件、类型驱动开发的新方法。传统上，类型被视为检查错误的工具，程序员首先编写一个完整的程序，然后使用编译器或运行时系统来检测类型错误。在类型驱动开发中，我们使用类型作为构建程序的工具。我们把类型放在首位，把它当作程序的计划，并使用编译器和类型检查器作为我们的助手，指导我们完成一个满足类型的完整的工作程序。我们预先放弃的类型越有表现力，我们就越有信心生成的程序是正确的。

类型和测试 “类型驱动开发”这个名称暗示了对测试驱动开发的类比。有一个相似之处，首先编写测试有助于确定程序的目的以及它是否满足一些基本要求。不同之处在于，与通常只能用于显示错误存在的测试不同，类型（使用得当）可以显示没有错误。但是，尽管类型减少了对测试的需求，但它们很少完全消除它。

Idris 是一种相对年轻的编程语言,从一开始就旨在支持类型驱动的开发。原型实现于 2008 年首次出现,当前实现的开发始于 2011 年。它建立在对编程语言和类型系统的理论和实践基础数十年的研究之上。

在 Idris 中,类型是一流的语言结构。类型可以像任何其他值(例如数字、字符串或列表)一样被操作、使用、作为参数传递给函数以及从函数返回。这是一个简单但强大的想法:

它允许表达值之间的关系;例如,两个列表具有相同的长度。

它允许编译器明确和检查假设。例如,如果您假设一个列表是非空的,Idris 可以确保在程序运行之前这个假设始终成立。如果需要,它允许程序行为被正式陈述并证明是正确的。

在本章中,我将介绍 Idris 编程语言并简要介绍它的特性和环境。我还将概述类型驱动的开发,讨论为什么类型在编程语言中很重要以及如何使用它们来指导软件开发。但首先,重要的是要准确理解我们谈论“类型”时的意思。

1.1 什么是类型?

我们从小就被教导识别和区分物体的类型。作为一个年幼的孩子,你可能有一个形状分类玩具。这包括一个盒子,顶部有各种形状的孔(见图 1.1)和一些可以穿过这些孔的形状。

有时他们配备了一个小塑料锤。这个想法是将每个形状(将其视为“值”)放入适当的孔(将其视为“类型”),可能通过锤子的强制。

在编程中,类型是对值进行分类的一种方式。例如,值 94、“thing”和[1,2,3,4,5]可以分别分类为整数、字符串和整数列表。就像你不能在形状分类器的圆孔中放入一个正方形一样,你不能在需要整数的程序部分使用像“thing”这样的字符串。

所有现代编程语言都按类型对值进行分类,尽管它们在何时以及如何这样做方面存在巨大差异(例如,它们是在编译时静态检查还是在运行时动态检查,类型之间的转换是否是自动的,等等)。

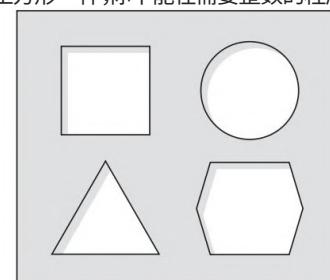


图 1.1 形状分类玩具的顶部。
形状对应于适合穿过孔的对象类型。

引入类型驱动开发

5

类型有几个重要的作用：

对于机器,类型描述了如何解释内存中的位模式。
 对于编译器或解释器,类型有助于确保解释位模式
 程序运行时始终如一。
 对于程序员来说,类型有助于命名和组织概念,帮助文档
 化和支持交互式编辑环境。

从本书的观点来看,类型最重要的目的是第三个。

类型以多种方式帮助程序员：

通过允许命名和组织概念(例如Square、
 圆形、三角形和六边形)
 通过提供变量、函数的用途的明确文档,
 和程序
 通过在交互式编辑环境中驱动代码完成

正如您将看到的,类型驱动开发特别广泛使用代码完成。尽管所有现代的静态类型语言都支持代码完成

在某种程度上,Idris 类型系统的表现力导致了强大的自动代码
 一代。

1.2 引入类型驱动开发

类型驱动开发是一种编程风格,我们首先编写类型,然后

使用这些类型来指导定义
 功能。整个过程就是写
 必要的数据类型,然后对每个函数执行以下操作:

- 1写出输入和输出类型。
- 2定义函数,使用输入类型的结构来指导
 执行。
- 3细化和编辑类型和功能
 必要时定义。

在类型驱动开发中,不是从检查的角度考虑类型,而是使
 用类型
 当你犯错时检查员会批评你,你可以把类型当成一个计
 划,
 使用类型检查器作为您的向导,
 引导您进入一个有效的、强大的程序。
 从一个类型和一个空函数开始
 身体,你逐渐向定义添加细节直到它完成,经常使用

类型作为模型

当你编写程序时,你会
 经常有一个概念模型
 你的头脑(或者,如果你有纪律,甚至在
 纸上)
 应该如何工作,组件如何交互,以及

数据是有组织的。这个模型是
 一开始可能很模糊
 并将变得更加精确
 程序不断发展,您的
 概念的理解
 发展。

类型允许您制作这些
 代码中明确的模型和
 确保您的实施
 程序匹配模型
 在你的脑海里。Idris 有一个表达型系
 统,允许您
 准确地描述模型
 根据您的需要,并细化
 模型与开发同时
 实施实施。

编译器检查程序到目前为止是否满足类型。伊德里斯,你很快就会看到,通过允许检查不完整的函数定义并提供用于描述类型的表达语言,强烈鼓励这种编程风格。

为了进一步说明,在本节中,我将展示一些示例来说明如何使用类型来详细描述程序打算做什么:矩阵算术,自动柜员机(ATM)建模,以及编写并发程序。那我就总结类型驱动开发的过程,介绍依赖类型,这将允许您表达程序的详细属性。

1.2.1 矩阵运算

矩阵是一个矩形的数字网格,按行和列排列。他们有几个科学应用程序,在编程方面,它们在密码学、3D图形、机器学习和数据分析中都有应用。以下为例,是一个 3×4 矩阵:

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$$

您可以在矩阵上实现各种算术运算,例如加法和乘法。要添加两个矩阵,请添加相应的元素,如您所见这里:

$$\begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix} + \begin{matrix} 7 & 8 \\ 9 & 10 \end{matrix} = \begin{matrix} 8 & 10 \\ 12 & 14 \\ 16 & 18 \end{matrix}$$

使用矩阵编程时,如果您从定义Matrix数据类型开始,则加法需要两个Matrix类型的输入,并给出一个Matrix类型的输出。但因为添加矩阵涉及添加输入的相应元素,所以如果两个输入具有不同的尺寸,会发生这种情况,就像这里一样?

$$\begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix} + \begin{matrix} 7 & 8 \\ 9 & 10 \end{matrix} = ? ? ?$$

如果您尝试添加不同维度的矩阵,那么您很可能已经在某处犯了错误。因此,您可以改进而不是使用Matrix类型

类型,使其包含矩阵的维度,并要求两个输入
矩阵具有相同的维度:

3×4 矩阵的第一个示例现在具有Matrix 3 4类型第一个 (正确) 加法示例采用Matrix .

3 2类型的两个输入和

给出Matrix 3 2类型的输出.

通过在矩阵类型中包含维度,您可以描述输入和

以这样的方式输出加法类型,即尝试添加不同大小的矩阵是类型错误。如果您尝试添加Matrix 3 2和Matrix 2 2,您的程序将不会

编译,更不用说运行了。

如果您在其类型中包含矩阵的维度,那么您需要考虑

每个矩阵的输入和输出维度之间的关系

手术。例如,转置矩阵涉及将行切换为列

反之亦然,所以如果你转置一个 3×2 矩阵,你最终会得到一个 2×3 矩阵:

1 2		
3 4	135	
转换为 ...	246	5 6

这个转置的输入类型是Matrix 3 2,输出类型是Matrix 2 3。

通常,我们不会在类型中给出确切的尺寸,而是使用变量来

描述输入的维度和维度之间的关系

输出。表 1.1 显示了输入维度和维度之间的关系

三个矩阵运算的输出:加法、乘法和转置。

表 1.1 矩阵运算的输入和输出类型。名称x、y 和
一般来说,描述输入和输出的维度是如何相关的。

手术	输入类型	输出类型
添加	矩阵 xy,矩阵 xy	矩阵 xy
乘	矩阵 xy,矩阵 yz	矩阵 xz
转置	矩阵 xy	矩阵 yx

我们将在第 3 章深入研究矩阵,在那里我们将详细介绍矩阵转置的实现。

1.2.2 自动柜员机

以及使用类型来描述输入和输出之间的关系

函数,与矩阵运算一样,您可以精确地描述运算何时进行

有效的。例如,如果您正在实现驱动ATM 的软件,您会想要

保证只有在用户输入卡后机器才会出钞,并且

验证了他们的个人识别码(PIN)。

要了解它是如何工作的,我们需要考虑ATM可能处于的状态: 准备就绪 ATM准备就绪并等待用户插入卡。

CardInserted ATM正在等待插入卡的用户进入

他们的PIN 码。

会话 验证会话正在进行中, ATM已验证用户的PIN,准备提取现金。

ATM 支持多种基本操作,每种操作仅在机器处于特定状态时才有效,并且每种操作都可能改变机器的状态,如图 1.2 所示。这些是基本操作:

InsertCard 等待用户插入卡片

EjectCard 从机器中弹出卡片 GetPIN 提示用户输入PIN

CheckPIN 检查输入的PIN是否正确

分配 分配现金

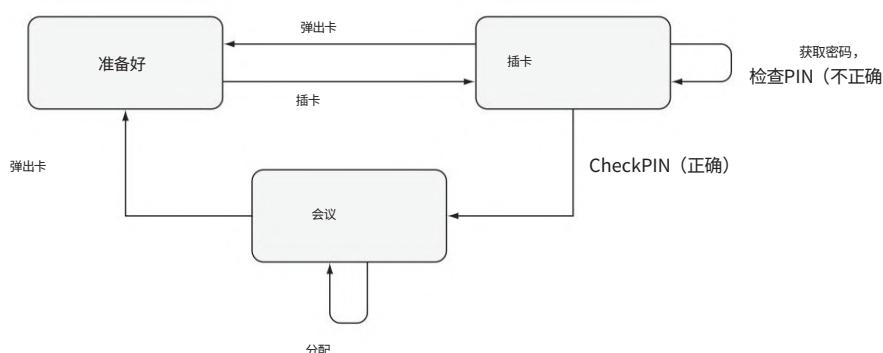


图 1.2 ATM 上的状态和有效操作。每个操作仅在特定状态下有效,并且可以改变机器的状态。仅当输入的 PIN 正确时, CheckPIN 才会更改状态。

操作是否有效取决于机器的状态。例如, InsertCard仅在Ready状态下有效,因为这是机器中唯一没有卡的状态。此外, Dispense仅在Session状态下有效,因为这是机器中唯一存在经过验证的卡的状态。

此外,执行这些操作之一可以改变机器的状态。例如, InsertCard将状态从Ready更改为CardInserted, CheckPIN 将状态从CardInserted更改为Session,前提是输入的 PIN 正确。

状态机和类型图 1.2 说明了一个状态机,描述了操作如何影响系统的整体状态。状态机通常隐含地存在于现实世界的系统中。例如,当您打开、阅读、

引入类型驱动开发

然后关闭一个文件,你用打开和关闭来改变文件的状态操作。正如你将在第 13 章中看到的,类型允许你创建这些状态显式更改,保证您仅在操作时才执行操作有效,并帮助您正确使用资源。

通过为ATM上的每个操作定义精确的类型,您可以保证,通过类型检查, ATM将只执行有效的操作。例如,如果您尝试实现一个无需验证PIN即可分发现金的程序,该程序将无法编译。通过在类型中显式定义有效的状态转换,您可以获得强大且机器可检查的关于其实现正确性的保证。我们将在第 13 章了解状态机,然后在第 14 章实现ATM示例。

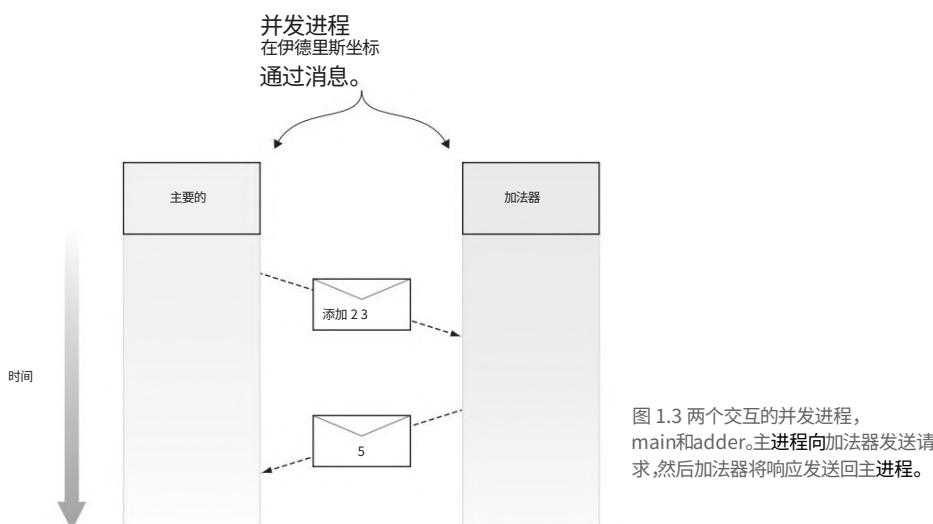
1.2.3 并发编程

并发程序由同时运行的多个进程组成,并且相互协调。并发程序可以响应并继续在运行大型计算时与用户交互。例如,用户可以下载大文件时继续浏览网页。此外,通过写并发程序,我们可以充分利用现代处理器的能力CPU,将工作分配给不同 CPU 内核上的多个进程。

在 Idris 中,进程通过发送和接收消息相互协调。

图 1.3 显示了一种可以工作的方式,有两个进程, main 和 adder。加法器进程等待来自其他进程的添加数字的请求。在它收到来自main的要求它添加两个数字的消息后,它会发送一个带有结果的响应。

然而,尽管有它的优点,并发编程是出了名的错误易于。进程相互交互的需求会大大增加系统的复杂性。对于每个进程,您需要确保它发送的消息和



接收与其他流程适当协调。例如,如果main和adder没有正确协调,并且每个都希望同时从另一个接收消息,它们就会死锁。

并发程序的类型与测试测试并发程序很困难,因为与纯顺序程序不同,不能保证来自不同进程的操作的执行顺序。即使在您运行一次测试时正确协调了两个进程,也不能保证在您下次运行测试时它们会正确协调。另一方面,如果你可以用类型来表达进程之间的协调,你就可以确定一个进行类型检查的并发程序有适当的协调进程。

当您编写并发程序时,理想情况下您将拥有一个流程应该如何交互的模型。使用类型,您可以使该模型在代码中显式化。然后,如果一个并发程序类型检查,你会知道它正确地遵循了模型。特别是,您可以做两件事:

为adder定义一个接口,描述它将处理的消息的形式。 定义一个协议,定义消息传递的顺序,确保main总是会向adder发送消息然后接收到回复,而adder总是会做相反的事情。

并发编程是一个广泛的主题,您可以通过多种方式使用类型来对进程之间的协调进行建模。我们将第 15 章看一个如何做到这一点的例子。

1.2.4 类型、定义、提炼:类型驱动开发的过程

在这些介绍性示例中的每一个中,我们都笼统地讨论了如何对系统进行建模:通过描述矩阵运算的输入和输出的有效形式、交互式系统的有效状态或消息在并发进程。在每种情况下,要实现系统,您首先要尝试找到一个捕获模型重要细节的类型,然后定义与该类型一起使用的函数,并根据需要细化该类型。

简而言之,您可以将类型驱动开发描述为类型、定义、细化的迭代过程:编写类型,实现满足该类型的函数,并在您对问题有更多了解时细化类型或定义。

例如,对于矩阵加法,您可以执行以下操作: 类型 - 编写一个 Matrix 数据类型,并将其用作加法函数的输入和输出类型。

定义 编写一个满足其输入和输出类型的加法函数。 **Refine** 注意加法函数的输入和输出类型允许你给出不同维度的无效输入,然后通过包含矩阵的维度使类型更精确。

通常,您将编写一个类型来表示您正在建模的系统,使用该类型定义函数,然后根据需要细化类型和定义以捕获任何缺失的属性。你会看到更多这种类型定义优化的过程在本书中,无论是在小规模实现单个功能时,在决定如何编写函数和数据类型时,规模更大。

1.2.5 依赖类型

在矩阵算术示例中,我们从Matrix类型开始,然后将其细化为包括行数和列数。例如,这意味着Matrix 3 4是 3×4 矩阵的类型。在这种类型中,3和4是普通值。依赖类型,例如Matrix,是一种根据其他值计算得出的类型。换句话说,它取决于其他值。

通过在这样的类型中包含值,您可以使类型尽可能精确。例如,一些语言有一个简单的列表类型,描述对象的列表。你可以通过对元素类型进行参数化使其更精确:一个通用列表字符串比简单列表更精确,并且不同于整数列表。你可以更精确的是依赖类型:4个字符串的列表与3个字符串的列表不同。

表 1.2 说明了 Idris 中的类型如何具有不同的精度级别,即使用于附加列表等基本操作。假设你有两个特定的输入字符串列表:

```
[ "A B C D" ]
[ "e" , "f" , "g" ]
```

附加它们时,您会看到以下输出列表:

```
[ "a" , "b" , "c" , "d" , "e" , "f" , "g" ]
```

使用简单类型,两个输入列表都具有AnyList类型,输出列表也是如此。用一个泛型类型,您可以指定输入列表都是字符串列表,输出也是列表。更精确的类型意味着,例如,输出显然与输入元素类型不变。最后,使用依赖类型,您可以指定输入和输出列表的大小。从类型可以看出长度输出列表是输入列表长度的总和。也就是说,3个字符串的列表附加到4个字符串的列表会产生7个字符串的列表。

表 1.2 附加特定类型的列表。与简单类型不同,它们之间没有区别输入和输出列表类型,依赖类型允许在类型中编码长度。

	输入 ["a" , "b" , "c" , "d"]	输入 ["e" , "f" , "g"]	输出类型
简单的	任意列表	任意列表	任意列表
通用的	列表字符串	列表字符串	列表字符串
依赖	Vect 4 字符串	Vect 3 字符串	Vect 7 字符串

列表和向量表 1.2 中类型的语法是有效的 Idris 语法。

Idris 提供了多种构建列表类型的方法,具有不同的精确度。在表格中,您可以看到其中的两个, List 和 Vect。AnyList 是包含在表中仅用于说明目的,未在表中定义伊德里斯。List 对没有明确长度的通用列表进行编码,而 Vect (简称“vector”) 使用类型中明确的长度对列表进行编码。你会看到很多本书中更多的这两种类型。

表 1.3 说明了如何在 Idris 中随着精度水平的提高而编写 10 个附加函数的输入和输出类型。使用简单类型,您可以编写输入输出类型为 AnyList,说明你对类型不感兴趣列表的元素。使用泛型类型,您可以将输入和输出类型编写为列出元素。这里, elem 是代表元素类型的类型变量。因为输入和输出的类型变量相同,类型指定两者输入列表和输出列表具有一致的元素类型。如果你附加两个整数列表,类型保证输出也将是整数列表。最后,使用依赖类型,您可以将输入写为 Vect n elem 和 Vect m elem,其中 n 和 m 是表示每个列表长度的变量。输出类型指定结果长度将是输入长度的总和。

表 1.3 一般附加类型列表。类型变量描述了它们之间的关系
输入和输出,即使确切的输入和输出是未知的。

	输入1种	输入2型	输出类型
简单的	任意列表	任意列表	任意列表
通用的	列出元素	列出元素	列出元素
依赖	向量	向量元素	Vect (n + m) 元素

类型变量 类型通常包含类型变量,如 n、m 和 elem

表 1.3。这些非常类似于 Java 或 C# 中泛型类型的参数,但它们在 Idris 中很常见,以至于它们具有非常轻量级的语法。在一般而言,具体类型名称以大写字母开头,类型变量名称以小写字母开头。

在表 1.3 的 append 函数的依赖类型中,参数 n 和 m 是普通数值, + 运算符是普通的加法运算符。所有的这些可以出现在程序中,就像它们出现在类型中一样。

入门练习

在本书中,练习将有助于巩固你所学的概念。作为一个热身,看看下面精选的功能规格,纯属给出以输入和输出类型的形式。对于他们每个人,建议可能的操作

这将满足给定的输入和输出类型。请注意,每种情况下可能有多个答案。

- 1 输入类型: Vect n elem
输出类型: Vect n elem
- 2 输入类型: Vect n elem
输出类型: Vect (n * 2) elem
- 3 输入类型: Vect (1 + n) elem
输出类型: Vect n elem
- 4 假设Bounded n表示一个介于零和n - 1 之间的数字。
输入类型: 有界n、Vect n elem
输出类型: elem

1.3 纯函数式编程

Idris 是一种纯函数式编程语言,因此在我们开始深入探索 Idris 之前,我们应该先了解一下函数式语言意味着什么,以及我们所说的纯度概念是什么意思。不幸的是,对于编程语言的功能性究竟意味着什么,并没有普遍认可的定义,但出于我们的目的,我们将其理解为以下含义:

程序由函数组成。程序执行包括函数的评估。

函数是一流的语言结构。

这与命令式编程语言的区别主要在于函数式编程关注函数的评估,而不是语句的执行。

在纯函数式语言中,以下内容也是正确的:

函数没有副作用,例如修改全局变量、抛出异常或执行控制台输入或输出。

因此,对于任何特定的输入,函数总是会给出相同的结果。

您可能非常合理地想知道,如何在这些限制下编写任何有用的软件。事实上,纯函数式编程不仅不会让编写实际程序变得更加困难,还可以让您以应有的尊重对待诸如状态和异常之类的棘手概念。让我们进一步探索。

1.3.1 纯度和参考透明度

纯函数的关键特性是相同的输入总是产生相同的结果。此属性称为参考透明度。如果函数中的表达式(例如函数调用)可以用其结果替换而不改变函数的行为,则它是引用透明的。如果函数只产生结果,没有副作用,那么这个属性显然是正确的。引用透明性在类型驱动开发中是一个非常有用的概念,因为如果一个函数没有副作用并且是



图 1.4 一个纯函数,接受输入并产生输出,没有可观察到的副作用

完全由其输入和输出定义,然后您可以查看它的输入和输出类型并清楚地了解函数可以做什么的限制。

图 1.4 显示了append函数的示例输入和输出。需要两个输入并产生结果,但没有与用户交互,例如阅读来自键盘,并且没有信息输出,例如日志记录或进度条。

图 1.5 显示了一般的纯函数。不可能有可观察的一面运行这些程序时的影响,除了可能使计算机稍微温暖或需要不同的时间来运行。



图 1.5 纯函数通常只接受输入并且没有可观察到的副作用。

纯函数在实践中非常普遍,尤其是在构造和操作数据结构时。可以对它们的行为进行推理,因为函数对于相同的输入总是给出相同的结果;这些功能很重要

大型程序的组成部分。前面的append函数是纯的,它是一个对于任何使用列表的程序来说都是有价值的组件。结果它产生一个列表,并且因为它是纯粹的,你知道它不需要任何输入、输出任何日志记录,或者做任何破坏性的事情,比如删除文件。

1.3.2 副作用程序

实际上,程序必须有副作用才能有用,而且你总是将不得不处理实际软件中的意外或错误输入。首先,这在纯语言中似乎是不可能的。不过有一种方法:纯函数可能无法执行副作用,但它们可以描述它们。

考虑一个从文件中读取两个列表、附加它们、打印结果列表并返回它的函数。下面的清单以命令式的形式概述了这个函数伪代码,使用简单类型。

清单 1.1 附加从文件中读取的列表 (伪代码)

```

列出 appendFromFile(File h) {
    list1 = readListFrom(h)
    list2 = readListFrom(h)

    结果 = 追加 (列表 1,列表 2)
    打印 (结果)
  
```

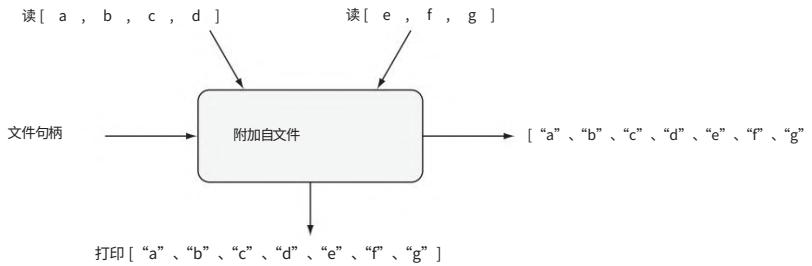


图 1.6 一个副作用程序,从文件中读取输入,打印结果,并返回结果

```

    返回结果
}

```

该程序将文件句柄作为输入,并返回一个带有一些副作用的列表。

它从给定文件中读取两个列表并在返回之前打印列表。图 1.6

当文件包含两个列表[a , b , c ,
d]和[e , f , g]。

`appendFromFile`函数不满足引用透明度属性。引用透明性要求表达式可以被其结果替换

不改变程序的行为。然而,在这里,用它的结果替换对`appendFromFile`的调用意味着不会从文件中读取任何内容,也不会

输出到屏幕上。函数的输入和输出类型告诉我们输入是文件,输出是一个列表,但类型中没有任何内容描述函数可能执行的副作用。

在一般的纯函数式编程中,尤其是在 Idris 中,您可以解决通过编写描述副作用的函数而不是描述副作用的函数来解决这个问题。执行它们,并将执行的细节推迟到编译器和运行时系统。我们将在第 5 章更详细地探讨这一点;现在,足以识别具有副作用的程序有一种类型可以明确说明这一点。例如,以下有区别:

字符串是产生字符串的程序类型,并保证不执行输入或输出作为副作用。
`IO String`是一种程序类型,它描述了一系列输入和输出操作,这些操作产生一个字符串。

类型驱动的开发将这个想法更进一步。正如你将在第 12 章中看到的那样之后,您可以定义描述程序可以产生的特定副作用的类型有,例如控制台交互,读写全局状态,或者产生并发进程和发送消息。

1.3.3 部分和全部功能

Idris 支持比纯函数更强大的属性,区分部分函数和全部函数。一个总函数保证产生一个

结果,这意味着它将在有限时间内为每个可能的良好类型返回一个值
输入,并且保证不会抛出任何异常。一个偏函数,在
另一方面,对于某些输入,可能不会返回结果。这里有几个例子:

append 函数对于有限列表是完全的,因为它总是返回一个新的
列表。

返回列表第一个元素的函数是部分的,因为它不是
如果列表为空,则定义,因此它将崩溃。

[总函数和长时间运行的程序](#)保证总函数产生一个可能无限结果的有限前缀。正如你将看到的

[第 11 章](#),您可以编写命令 shell 或服务器作为总函数,
无限期地保证对每个用户输入的响应。

区别很重要,因为知道函数是全部的,因此您可以
根据其类型对其行为做出更强有力的声明。如果你有一个功能
例如,返回类型为String 时,您可以根据不同的要求提出不同的声明
关于函数是部分的还是全部的。

如果是总计 它将在有限时间内返回一个字符串类型的值。

如果是部分的 如果它没有崩溃或进入无限循环,它返回的值将
成为一个字符串。

在大多数现代语言中,我们必须假设函数是部分的,因此只能使后者更弱,声称。Idris 检查函数是否是总
的,所以我们
因此,通常可以使前者更强大,声称。

总功能和停机问题

停止问题是确定程序是否终止的问题

一些特定的输入。感谢 Alan Turing,我们知道不可能写出

一般解决停机问题的程序。鉴于此,有理由想知道 Idris 如何确定一个函数是完全的,这本质上是检查

一个函数对所有输入都终止。

虽然它不能解决一般问题,但 Idris 可以识别出一大类绝对是总的函数。您将了解更多关于它是如何做到这一点
的,以及一些

编写总函数的技巧,第 10 章和第 11 章。

类型驱动开发中一个有用的模式是编写一个精确描述的类型
系统的有效状态 (如第 1.2.2 节中的ATM)并限制操作

系统被允许执行。然后,类型检查器保证具有该类型的总函数能够按照类型要求精确地执行这些操作。

1.4 伊德里斯快速浏览

Idris 系统由一个交互式环境和一个批处理模式编译器组成。在交互式环境,您可以加载和类型检查源文件,评估表达式,搜索库,浏览文档,编译运行完成程式。我们将在本书中广泛使用这些特性。

在本节中,我将简要介绍环境中最重要的特性,即评估和类型检查,并描述如何编译和运行

伊德里斯程序。我还将介绍 Idris 语言本身最显着的两个特征:

孔,代表不完整的程序
使用类型作为一流的语言结构

正如您将看到的,通过使用孔,您可以增量定义函数,询问类型
检查上下文信息以帮助完成定义。使用一流
类型,您可以非常精确地了解函数的用途,甚至可以询问
类型检查器为您填写功能的一些细节。

1.4.1 交互环境

您与 Idris 的大部分互动将通过一个名为
read-eval-print 循环,通常缩写为REPL。顾名思义, REPL
将从用户那里读取输入,通常以表达式的形式,计算表达式,然后打印结果。

安装 Idris 后,您可以通过在 shell 提示符下键入 idris 来启动REPL。
您应该会看到如下内容:

```
_____
/_/_//__(_)
///_/_/_/_//_/_/(_)/_/
\_,/_/_/_/
1.0 版
http://www.idris-lang.org/
类型 : ?求助
```

Idris 是完全没有保证的免费软件。
有关详细信息,请键入 :保修。
伊德里斯>

安装 IDRIS 您可以找到有关如何下载和安装的说明
附录 A 中适用于 Linux、OS X 或 Windows 的 Idris。

您可以在Idris>提示符下输入要计算的表达式。例如,算术元表达式以常规方式工作,具有通常的优先规则 (即
也就是说, *和/的优先级高于+和-):

伊德里斯>2+2
4:整数

```
伊德里斯> 2.1 * 20
42.0:Double
```

```
伊德里斯> 6 + 8 * 11
94:Int
```

您还可以操作字符串。 ++运算符连接字符串， reverse函数反转字符串：

```
伊德里斯> "你好" ++
           ++ "世界！"
"你好世界！" : String

Idris> 反向 "abcdefg" "gfedcba" :String
```

请注意,Idris 不仅打印出表达式的计算结果,还打印出它的类型。一般来说,如果你看到 $x : T$ 形式的东西
某个表达式 x 、一个冒号和某个其他表达式 T 这可以被解读为“ x 具有类型 T ”。在前面的示例中,您有以下
内容： 4具有Integer 类型。 42.0具有Double 类型。 “你好,世界！”具有字符串类型。

1.4.2 检查类型

REPL提供了许多命令,所有命令都以冒号为前缀。最常用的一种是: $:t$,它允许您检查表达式的类型,而无需对
它们求值：

```
伊德里斯> :t 2 + 2
2+2:Int

伊德里斯> :t "你好！"
"你好！" : String
```

类型,例如Integer和String,可以像任何其他值一样被操作,因此您也可以检查它们的类型：

```
伊德里斯> :t 整数
整数 : Type

伊德里斯> :t 字符串
字符串 : Type
```

很自然地想知道Type本身的类型可能是什么。在实践中,您永远不需要担心这一点,但为了完整起见,让我们
看一下：

```
伊德里斯> :t Type
Type : Type 1
```

也就是说,就我们而言, Type具有类型1,类型 1具有类型2,以此类推。好消息是 Idris 会为你处理细节,你总是可以单独编写Type。

1.4.3 编译和运行 Idris 程序

除了评估表达式和检查函数类型之外，您还希望能够编译和运行完整的程序。下面的清单显示了一个最小的 Idris 程序。

清单 1.2 你好,伊德里斯世界! (你好.idr)

```
模块主要                                ↪ 模块头  
main : IO () main =  
putStrLn "Hello, Idris World!"          ↪ 函数声明  
                                         ↪ 函数定义
```

在这个阶段，无需过多担心语法或程序的工作方式。现在，您只需要知道 Idris 源文件由模块头和函数和数据类型定义的集合组成。他们还可以导入其他源文件。

WHITESPACE SIGNIFICANCE 空白在 Idris 中很重要, 所以当你键入清单 1.2 时, 请确保每行的开头没有空格。

在这里,该模块称为Main,并且只有一个函数定义,称为main。

任何 Idris 程序的入口点都是 Main 模块中的 main 函数。

要运行该程序,请按照下列步骤操作: 1在文本

编辑器中创建一个名为 Hello.idr 的文件。1 Idris 源文件都有扩展名 .idr

2 输入清单 1-2 中的代码。

3在保存 Hello.idr 的工作目录中,使用以下命令启动 Idris REPL 命令 idris Hello.idr。

4 在 Idris 提示符下, 键入 :exec。

如果一切顺利,您应该会看到如下内容:

```
$ idris Hello.idr
```

1.0 版 <http://www.idris-lang.org/> 类型：求助

Idris 是完全没有保证的免费软件。

有关详细信息,请键入“保修”。

类型检查 ./Hello.idr

¹ 我推荐 Atom，因为它有一个 Idris 程序的交互式编辑模式，我们将在本书中使用它。

```
*你好> :exec
你好,伊德里斯世界
```

在这里, \$代表你的 shell 提示符。或者,您可以通过调用带有-o选项的idris命令来创建一个独立的可执行文件,如下所示:

```
$ idris Hello.idr -o 你好
$ ./你好
你好,伊德里斯世界
```

REPL提示符 默认情况下, REPL提示符会告诉您当前加载的文件。 Idris >提示符表示没有文件已加载,而提示符*Hello>表示 Hello.idr 文件是加载。

1.4.4 不完整的定义:使用孔

之前,我将使用类型和值与将形状插入形状分类器玩具进行了比较。就像正方形只能穿过一个方孔一样,这个论点

“你好,伊德里斯世界!”仅适合String类型所在位置的函数预期的。

Idris 函数本身可以包含孔,带有孔的函数是不完整的。只有适当类型的值才能放入孔中,就像正方形一样

只能装入形状分类器的方孔中。这是 “Hello, Idris World!”的不完整实现。程序:

```
模块主要
主要:我 ()
main = putStrLn ?问候
```



?greeting 是一个洞,代表程序中缺少的部分。

如果您编辑 Hello.idr 以替换字符串 “Hello, Idris World!”带? 问候和将其加载到 Idris REPL 中,您应该会看到如下内容:

```
类型检查 ./Hello.idr
漏洞:Main.greeting
*你好>
```

语法?greeting引入了一个漏洞,它是尚未编写的程序的一部分 10。您可以对带有孔的程序进行类型检查并在REPL上对其进行评估。

在这里,当 Idris 遇到?greeting孔时,它会创建一个新名称, greeting,有类型但没有定义。您可以在REPL中使用:t检查类型:

```
*你好> :t 问候
-----
问候语:字符串
```

另一方面,如果您尝试评估它,伊德里斯会告诉您这是一个洞:

```
*你好>问候
? 问候:字符串
```

重新加载除

了退出 REPL 并重新启动之外,您还可以使用:r REPL 命令重新加载 Hello.idr,如下所示:

```
*Hello> :r 类型检
查 ./Hello.idr Holes: Main.greeting *Hello>
```

孔允许您逐步开发程序,编写您知道的部分并要求机器通过识别您不知道的部分的类型来帮助您。例如,假设您想打印一个字符 (类型为Char)而不是字符串。 putStrLn函数需要一个String参数,因此您不能简单地将Char传递给它。

清单 1.3 一个类型错误的程序

模块主要

```
main : IO () main =
putStrLn x
```

输入错误,给出一个字
符而不是一个字符串

如果你尝试将此程序加载到REPL 中, Idris 会报错:

```
Hello.idr:4:17 检查 main 右侧时:检查函数 Prelude.putStrLn 的应用程序时:Char 之间的类型不匹配 ( “x”
类型)
```

和

字符串 (预期类型)

您必须以某种方式将Char转换为String。即使您一开始并不确切知道如何执行此操作,您也可以先添加一个孔来代替转换。

模块主要

```
main : IO () main =
putStrLn (?convert x )
```

然后你可以检查转换孔的类型:

```
*你好> :t 转换
-----
转换:字符 -> 字符串
```

这是一个函数类型,以一个 Char 作为输入并返回一个 String。

洞的类型, Char -> String,是一个函数的类型,它接受一个Char作为输入并返回一个String作为输出。我们将在第 2 章更详细地讨论类型转换,但完成这个定义的适当函数是强制转换:

```
main : IO () main =
putStrLn (cast x )
```

1.4.5 头等类型

一流的语言结构是一种被视为一个值的结构,它的使用位置没有语法限制。换句话说,第一类构造可以传递给函数、从函数返回、存储在变量中等等。

在大多数静态类型语言中,类型的使用位置受到限制,类型和值之间存在严格的句法分离。例如,您不能在 Java 方法或 C 函数的主体中说`x = int`。在 Idris 中,没有这样的限制,类型是一流的;不仅可以以与任何其他语言构造相同的方式使用类型,而且任何构造都可以作为类型的一部分出现。

这意味着您可以编写计算类型的函数,并且函数的返回类型可以根据函数的输入值而有所不同。在 Idris 中编程时经常会出现这个想法,并且有几种实际情况很有用:

数据库模式决定了数据库上允许的查询形式。 网页上的表单决定了预期输入的数量和类型。

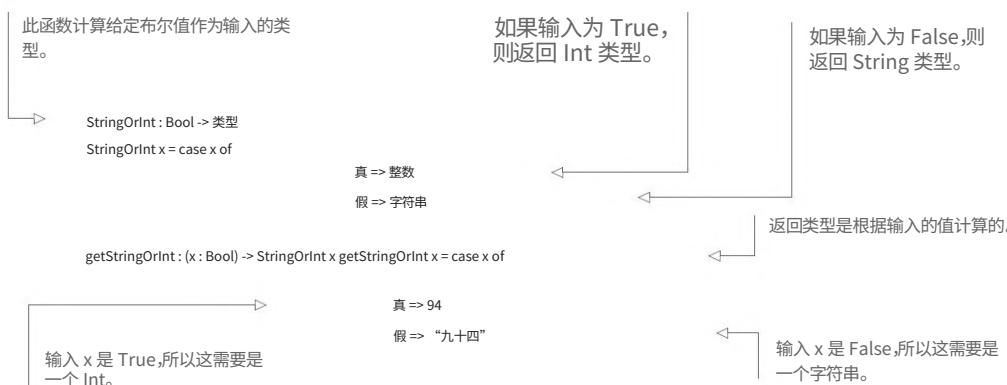
网络协议描述决定了可以发送的值的类型

或通过网络接收。

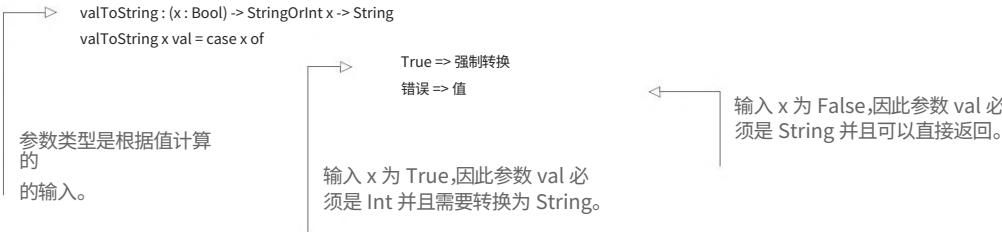
在每种情况下,一条数据都会告诉您一些其他数据的预期形式。如果您使用 C 进行编程,您会看到`printf`函数的类似想法,其中一个参数是一个格式字符串,它描述了剩余参数的数量和预期类型。C 类型系统无法检查格式字符串是否与参数一致,因此这种检查通常被硬编码到 C 编译器中。然而,在 Idris 中,您可以直接编写类似于`printf`的函数,利用类型作为一等构造。你会在第 6 章看到这个具体的例子。

以下清单通过一个小示例说明了第一类类型的概念:从布尔输入计算类型。

清单 1.4 计算一个类型,给定一个布尔值 (FCTypes.idr)



伊德里斯快速浏览

**函数语法**

我们将在接下来的章节中更详细地介绍 Idris 语法。就目前而言
请记住以下几点：

函数类型采用 $a \rightarrow b \rightarrow \dots \rightarrow t$ 的形式，其中 a 、 b 等，
是输入类型， t 是输出类型。输入也可以被注释
带有名称，形式为 $(x : a) \rightarrow (y : b) \rightarrow \dots \rightarrow name : type$ 声明一个类型为
 $type$ 的新函数 $name$ 。

函数由方程定义：

$\text{平方} = x * x$

这定义了一个名为 square 的函数，该函数将其输入乘以自身。

这里， StringOrInt 是一个计算类型的函数。清单 1.4 以两种方式使用它：

在 getStringOrInt 中， StringOrInt 计算返回类型。如果输入是
是的， getStringOrInt 返回一个 Int ；否则返回一个字符串。

在 valToString 中， StringOrInt 计算一个参数类型。如果第一个输入是
确实，第二个输入必须是 Int ；否则它必须是一个字符串。

你可以通过在 valTo 的定义中引入漏洞来详细了解发生了什么
细绳：

```

valToString : (x : Bool) -> StringOrInt x -> String
valToString x val = case x of
  真 => ?xtrueType
  假 => ?xfalseType
  
```

使用 $:t$ 检查孔的类型不仅可以提供孔本身的类型，还可以提供
还有范围内任何局部变量的类型。如果您检查 $xtrueType$ 的类型，您将
查看 val 的类型，它是在已知 x 为 True 时计算的：

```

*FCTypes> :t xtrueType
x:布尔
值:整数
-----
```

$xtrueType$: 字符串

因此,如果x为True,则val必须是Int,由StringOrInt函数计算得出。同样的,你可以检查xfalseType的类型来查看当x已知时val的类型为假:

```
*FCTypes> :t xfalseType
x:布尔
值:字符串
-----
xfalseType : 字符串
```

这是一个小例子,但它说明了类型驱动开发和依赖类型编程的一个基本概念:变量的类型

可以从另一个值计算。在每种情况下,Idris都使用了StringOrInt根据它对x值的了解来改进val的类型。

1.5 总结

类型是对值进行分类的一种手段。编程语言使用类型来决定如何在内存中布局数据,并确保数据被解释始终如一。

可以将类型视为规范,因此语言实现(特别是其类型检查器)可以检查程序是否符合该规范。

类型驱动开发是类型、定义、提炼、创建的迭代过程
一个类型来建模一个系统,然后定义函数,最后细化类型
有必要的。
在类型驱动的开发中,类型更像是一个计划,帮助一个交互
活动环境将程序员引导到工作程序。
依赖类型允许你给程序提供更精确的类型,因此
更多信息的机器计划。
在函数式编程语言中,程序执行包括评估
荷兰国际集团的功能。
在纯函数式编程语言中,另外,函数没有
副作用。
与其编写执行副作用的程序,不如编写程序
描述副作用的,在程序的
类型。
总函数保证为任何类型良好的输入产生结果
有限的时间。
Idris是一种专门为支持类型驱动开发而设计的编程语言。它是一种纯函数式编程语言,具有一流的
依赖类型。

Idris允许程序包含代表不完整程序的孔。
在Idris中,类型是一等的,这意味着它们可以存储在变量中,通过
到函数,或者像任何其他值一样从函数返回。

伊德里斯入门



本章涵盖

使用内置类型和函数

定义功能

构建 Idris 程序

在学习任何新语言时,重要的是要牢牢掌握基础知识,然后再学习该语言的更显着特征。和

记住这一点,在我们开始探索依赖类型和类型驱动开发本身之前,我们将看看一些你熟悉的类型和值

其他语言,您将看到它们在 Idris 中是如何工作的。您还将看到如何定义函数并将它们组合在一起以构建一个完整的,即使简单的 Idris 程序。

如果你已经熟悉一种纯函数式语言,尤其是 Haskell,
本章的大部分内容看起来非常熟悉。清单 2.1 展示了一个简单但自包含的 Idris 程序,它反复提示来自控制台的输入和
然后显示输入中单词的平均长度。如果您已经能够在注释的帮助下轻松阅读该程序,则可以
安全地跳过此
章,因为它故意避免引入任何特定于 Idris.¹ 的语言特性

¹ 将 Idris 与 Haskell 进行比较,最重要的区别是 Idris 不使用惰性求值默认。

即便如此,我仍然建议您浏览本章的提示和注意事项并阅读最后总结,以确保您没有遗漏任何小细节。

否则,别担心。在本章结束时,我们将涵盖所有的 nec 基本功能让您能够自己实现类似的程序。

清单 2.1 一个完整的 Idris 计算平均字长的程序 (Average.idr)

```

所有顶级函数都必须有类型声明。
作为类型声明的一部分,可以选择为参数类型指定名称。在这里,有一个类型为 String 且名称为 str 的参数。
                                         cast 函数显式转换
                                         类型之间。在这里,除法运算符
                                         需要一个 Double,但总长度和
                                         numWords 变量是 Nats。
                                         ↓
模块主要
平均 : (str : String) -> Double
平均 str = 让 numWords = wordCount str
    totalLength = sum (allLengths (words str)) in
    投射总长度 / 投射 numWords
                                         ↓
在哪里
wordCount : 字符串 -> Nat
wordCount str = 长度 (单词 str)
                                         ↓
allLengths : 列表字符串 -> 列表 Nat
allLengths strs = 地图长度 strs
                                         ↓
showAverage : 字符串 -> 字符串
showAverage str = 平均字长为:
    显示 (平均 str)++ "\n"
                                         ↓
主要:我 ()
main = repl 输入一个字符串: showAverage
                                         ↓
函数 main,在一个名为 Main 的模块
中,是 Idris 程序的入口点。
                                         ↓
repl 是一个重复显示提示的函数,从控
制台读取一个字符串,然后
显示对该字符串运行函数的结果。
                                         ↓
++
```

字符串是一种原始类型,不像其他一些语言(特别是 Haskell,其中字符串表示为字符列表)。

2.1 基本类型

Idris 提供了一些标准的基本类型和函数来处理各种形式,字符和字符串。在本节中,我将概述这些,以及一些例子。这些基本类型在 Prelude 中定义,它是一个集合每个 Idris 程序自动导入的标准类型和功能。

我将在本节中向您展示几个示例表达式,它们可能看起来相当明确他们应该做什么。然而,与其简单地阅读它们并点头,我强烈建议您在 Idris REPL 上键入示例。通过使用它,您将比仅阅读它更容易地学习语法税。

在此过程中,我们还将遇到一些有用的REPL功能,这些功能将允许我们将计算结果存储在REPL 中。

THE PRELUDE 我将讨论的类型和函数在 Prelude 中定义。

Prelude 是 Idris 的标准库，在 REPL 中始终可用，并由每个 Idris 程序自动导入。除了一些原始类型和操作之外，Prelude 中的所有内容都是用 Idris 本身编写的。

2.1.1 数值类型和值

Idris 提供了几种基本的数字类型，包括：

Int 固定宽度的有符号整数类型。它保证至少 31 位宽，但确切的宽度取决于系统。

Integer 无界有符号整数类型。与 Int 不同，除了你机器的内存之外，可以表示的数字的大小没有限制，但是这种类型在性能和内存方面更昂贵

用法。

Nat 无界无符号整数类型。这通常用于数据结构的大小和索引，因为它们永远不会是负数。稍后你会看到更多关于 Nat 的内容。

Double 双精度浮点类型。

NATS 减法 因为 Nat 永远不会是负数，所以 Nat 只能从较大的 Nat 中减去。

我们可以使用标准数字文字作为每种类型的值。例如，文字 333 可以是 Int、 Integer、 Nat 或 Double 类型。由于显式小数点，文字 333.0 只能是 Double 类型。

你可以在 REPL 上尝试一些简单的计算：

```
伊德里斯> 6 + 3 * 12
42:整数伊德里斯> 6.0 +
3 * 12
42.0:双倍
```

请注意，Idris 会将数字视为

默认情况下为整数，除非有一些上下文，并且两个操作数必须是相同的类型。因此，在前面两个表达式的第二个中，文字 6.0 只能是一个 Double，所以整个表达式是一个 Double，3 和 12 也被视为 Double。

当表达式（例如 $6 + 3 * 12$ ）可以是多种类型之一时，您可以使用 `<type><expression>` 表示法明确表示该类型，表示该类型是所需的表达式类型：

REPL 结果

REPL 的最新结果总是可以通过使用它的特殊值来检索并用于进一步的计算：

```
伊德里斯> 6 + 3 * 12
```

```
42:整数
伊德里斯> 它 * 2
84:整数
```

也可以使用 `:let` 命令将表达式绑定到 REPL 中的名称：

```
伊德里斯> :let x = 100
伊德里斯> x
100 : 整数 Idris> :let y =
200.0 Idris> y 200.0 : 双倍
```

```
伊德里斯> 6 + 3 * 12
42:整数
伊德里斯> Int (6 + 3 * 12)
42:国际
伊德里斯> 双重 (6 + 3 * 12)
42.0:双倍
```

“THE” EXPRESSIONS不是内置语法,而是一个普通的 Idris 函数,在 Prelude 中定义,它利用了一流的类型。

2.1.2 使用强制转换的类型转换

算术运算符适用于任何数字类型,但输入和输出必须具有相同的类型。因此,有时您需要在类型之间进行转换。

假设您在REPL中定义了一个Integer和一个Double：

```
Idris> :let integerval =6*6 Idris> :let doubleval =0.1
伊德里斯>整数
36:整数
伊德里斯>双倍
0.1:双倍
```

如果您尝试添加integerval和doubleval, Idris 会抱怨它们不是同一类型：

```
Idris> integerval + doubleval (input):1:8-9:当检查函数
Prelude.Classes.+的应用时:Double (Type of doubleval)之间的类型不匹配
```

和
整数 (预期类型)

要解决此问题,您可以使用cast函数,该函数将其输入转换为所需的类型,只要该转换有效。在这里,您可以将Integer转换为Double：

```
Idris> cast integerval + doubleval 36.1 : Double
```

Idris 支持所有原始类型之间的转换,并且可以添加用户定义的转换,正如您将在第 7 章中看到的那样。请注意,某些转换可能会丢失信息,例如将Double转换为Integer。

指定演员表的目标

您还可以使用指定要转换为的类型,如以下示例所示：

```
伊德里斯> 整数 (施放 9.9)
9:整数
伊德里斯> 双重 (施法 (4 + 4) )
8.0:双倍
```

2.1.3 字符和字符串

Idris 还提供 Unicode 字符和字符串作为原始类型,以及一些有用的原始函数来操作它们。字符文字 (Char 类型)用单引号括起来,例如

```
一个 。  
字符串字面量 (String 类型) 用双引号括起来, 例如  
“你好世界！”
```

与许多其他语言一样,Idris 通过使用以反斜杠开头的转义序列来支持字符和字符串文字中的特殊字符。例如,换行符使用\n 表示:

```
伊德里斯> :t \n \n :  
字符  
伊德里斯> :t 你好世界!\n :字符串  
Hello world\n :字符串
```

这些是最常见的转义序列:

\ 用于文字单引号	\ 用于文
字双引号	\\用于文字反斜杠
\n用	\n用
于换行符	\t用于制表符

Prelude 定义了几个有用的函数来操作字符串。您可以在REPL中看到其中的一些实际操作:

```
伊德里斯> length "你好！"  
6: 纳特  
伊德里斯> reverse "抽屉"  
“奖励” :字符串  
伊德里斯> substr 6 5 "你好世界"  
“世界” :字符串  
伊德里斯> "你好"++ "世界"  
“你好世界” :字符串
```

以下是对这些函数的简要说明: length - 将其参数

的长度作为Nat给出,因为String不能有
负长度 reverse
- 返回其输入的反转版本 substr - 返回输入字符串的子字符串
串,给定起始位置和所需的子字符串长度 ++ - 连接两个字符串的运算符

注意函数调用的语法。在 Idris 中,函数与它们的函数是分开的空格的参数。如果参数是复杂表达式,则必须用括号括起来,如下所示:

```
伊德里斯>长度 ( “你好” ++
++ “世界” )
11:纳特
```

[函数语法](#)通过用空格分隔参数来调用函数

起初可能看起来很奇怪。不过,这是有充分理由的,当我们在本章后面看函数类型时你会发现。简而言之,它使操作功能更加灵活。

2.1.4 布尔值

Idris 提供了一个Bool类型来表示真值。 Bool可以取值真或假。运算符&&和||分别表示逻辑和和或:

```
伊德里斯> 真 && 假
假:布尔值
伊德里斯> 真||错误的
真:布尔
```

常用的比较运算符 (<、 <=、 ==、 /=、 >、 >=)可用:

```
伊德里斯>3>2
真:布尔
伊德里斯> 100 == 99
假:布尔值
伊德里斯> 100 /= 99
真:布尔
```

不等式Idris中的不等式运算符是/=,它遵循 Haskell 语法,而不是!=,它遵循 C 和 Java 等语言的语法。

还有一个if...then...else结构。这是一个表达式,所以它必须始终包括then分支和else分支。例如,你可以写一个表达式,根据长度计算为不同的消息作为字符串一句话:

```
伊德里斯> :let word = 编程
Idris> if length word > 10 then 多么长的词啊! 其他 “短字”
“好长的一句话!” : 细绳
```

2.2 函数:Idris 程序的构建块

现在您已经了解了一些基本类型和简单的控制结构,您可以开始定义功能。在本节中,您将使用基本函数编写一些 Idris 函数到目前为止您已经看到的类型,将它们加载到 Idris 系统中,并在REPL 中对其进行测试。您还将看到函数式编程风格如何让您编写更通用的程序有两种方式:

在函数类型中使用变量,以便可以编写函数来处理几种不同的类型。使用高阶函数来捕获常见的编程模式。

2.2.1 函数类型和定义

函数类型由一种或多种输入类型和输出类型组成。例如,将一个Int作为输入并返回另一个Int的函数将被写为`Int -> Int`。下面的清单显示了一个具有这种类型的简单函数定义,即`double`函数。

清单 2.2 将Int加倍的函数(Double.idr)

```
→ 双 : Int -> Int
    双倍=x+x ←
函数类型声明它接受一个 Int 作为输入并返回
一个 Int 作为输出。
```

函数定义给出了一个方程,该方程定义了将输入加倍的含义。

您可以通过将其键入文件 `Double.idr` 来尝试此功能;通过在 shell 提示符下键入 `idris Double.idr` 将其加载到 Idris REPL 中;然后在REPL 上尝试一些示例:

```
*双>双47
94:国际

*双>双 (双15)
60:整数
```

图 2.1 显示了这个函数定义的组件。Idris 中的所有函数,如`double`,都是通过类型声明引入的,然后由具有左侧和右侧的方程定义。

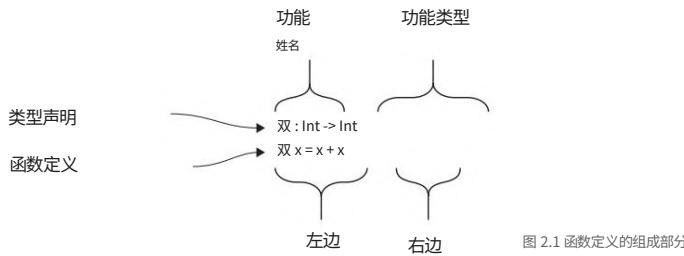


图 2.1 函数定义的组成部分

通过根据这些等式重写表达式来评估表达式,直到无法进行进一步的重写。因此,函数定义了可以重写表达式的规则。例如,考虑`double` 的定义:

`双倍=x+x`

这意味着每当 Idris 求值器遇到一个形式为`double x` 的表达式,其中某个表达式代表`x`,它应该被重写为`x + x`。

因此,在示例`double (double 15)` 中,首先,内部`double`

`15`被重写为`15 + 15`。`15 + 15`被重写为`30`。

`double 30`被重写为`30 + 30`。

`30 + 30`被改写为`60`。

评估顺序您可能已经

注意到,与其选择先评估内部双 15 ,不如选择外部双 (双 15) ,这将减少为双 15 + 双 15。任何一种顺序都是可能的,并且每个都会领先到同样的结果。默认情况下,Idris 将首先计算最里面的表达式。换句话说,它将在函数定义之前评估函数参数。

两种选择都各有优缺点,因此这个话题一直在争论不休!现在不是重新讨论这个辩论的时候,但如果你有兴趣,你可以研究一下惰性评估。 Idris 支持使用显式类型惰性求值,您将在第 11 章中看到。

或者,您可以在函数的输入类型中给出明确的名称。例如,您可以编写`double`的类型,如下所示:

双: (值:Int) -> Int

这与前面的声明具有完全相同的含义 (`double : Int -> Int`) 。您可能会明确说明参数的名称有两个原因:

在类型中命名参数可以给读者一些关于

论证的目的。

为参数命名意味着您可以稍后引用它。

当我们开始深入探索依赖类型时,你会在第 4 章看到更多这方面的内容。现在,请记住第 1 章中一等类型的示例,其中我为`getStringOrInt` 提供了以下 Idris 类型:

```
getStringOrInt : (x : Bool) -> StringOrInt x
```

第一个`Bool`类型的参数被命名为`x`,然后出现在返回类型中。

类型声明是必需的! Idris 中的函数必须有明确的类型声明,例如`double : Int -> Int`这里。其他一些函数式语言,尤其是 Haskell 和 ML,允许程序员省略类型声明并让编译器推断类型。然而,在具有一流类型的语言中,这通常证明是不可能的。无论如何,这是

功能:Idris 程序的构建块

在类型驱动开发中省略类型声明是不可取的。我们的哲学是使用类型来帮助我们编写程序,而不是使用程序
帮助我们推断类型!

2.2.2 部分应用函数

当一个函数有多个参数时,你可以创建一个专门的版本
函数通过省略后面的参数。这称为部分应用。

例如,假设您有一个将两个整数相加的add函数,定义为
在 Partial.idr 文件中如下:

```
添加: Int -> Int -> Int
添加 xy = x + y
```

如果将函数应用于两个参数,它将评估为Int:

```
*部分>加2 3
5:诠释
```

另一方面,如果你只将函数应用于一个参数,省略第二个参数,Idris 将返回一个Int -> Int 类型的函数:

```
*部分>加2
添加 2 : Int -> Int
```

通过仅对一个参数应用add ,您创建了一个新的专用函数,添加
2,它的参数增加了2 。您可以通过使用:let创建一个新函数来更明确地看到这一点:

```
*Partial> :let addTwo = 添加 2
*部分> :t addTwo
addTwo : Int -> Int
*部分> addTwo 3
5:诠释
```

函数应用语法,简单地通过用空格将函数与参数分开来将函数应用于参数,为部分应用提供了特别简洁的语
法。部分应用程序在 Idris 程序中很常见,您将

请参阅第 2.2.5 节中的一些示例。

2.2.3 编写泛型函数 :类型中的变量

与具体类型 (例如Int、 String和Bool)一样,函数类型也可以包含变量。函数类型中的变量可以用不同的值
实例化,就像
函数本身的变量。

例如,让我们考虑返回其输入的恒等函数,
不变。 Ints上的恒等函数写法如下:

```
identityInt : Int -> Int
身份整数 x = x
```

类似地， Strings上的恒等函数是这样写的：

```
身份字符串:字符串->字符串身份字符串 x = x
```

这是布尔的恒等函数：

```
identityBool : Bool -> Bool identityBool x = x
```

您可能已经注意到这里的一个模式。在每种情况下，定义都是相同的！您无需了解有关x的任何信息，因为您在每种情况下都将其原封不动地返回。因此，您可以在类型级别使用变量ty代替具体类型，而不是分别为每种类型编写一个恒等函数：

```
身份 :ty -> ty 身份x=x
```

[ID 函数](#)实际上，Prelude 中有一个名为id 的标识函数，其定义方式与此处的标识完全相同。

标识类型中的ty是一个变量，代表任何类型。因此，可以使用任何输入类型调用identity，并将返回与输入具有相同类型的输出。

类型中的变量名称任何出现在类型声明中的名称，以小写字母开头，否则未定义的名称被假定为变量。请注意，我很小心地调用这些变量，而不是类型变量。

这是因为，对于依赖类型，类型中的变量不一定只代表类型，正如您将在第 3 章中看到的那样。

在处理数字类型时，您已经看到了恒等函数的一种形式：是恒等函数。它在 Prelude 中定义如下：

```
the : (ty : Type) -> ty -> ty ty x = x
```

它将显式类型作为其第一个参数，显式命名为ty。第二个参数的类型由第一个参数的输入值给出。这是一个实际依赖类型的简单示例，因为前面的参数的值给出了后面的参数的类型。您可以在REPL 中明确地看到这一点，方法是将部分应用于仅一个参数：

```
伊德里斯> :t int
Int : Int -> Int

Idris> :t 字符串 String : String -> String
```

```
伊德里斯> :t 布尔值
布尔值 :布尔值 -> 布尔值
```

2.2.4 编写具有约束类型的泛型函数

您在第 2.2.1 节中看到的第一个函数double将作为输入给出的Int加倍：

```
双 : Int -> Int
双倍=x+x
```

但是其他数字类型呢？例如，您还可以编写一个函数来将Nat或Integer 加倍：

```
双纳特 : Nat -> Nat
双纳特 x=x+x

doubleInteger : Integer -> Integer doubleInteger x=x+x
```

与身份一样，您可能开始在这里看到一种模式，所以让我们看看如果我们尝试用变量替换输入和输出类型会发生什么。将以下内容放入名为 Generic.idr 的文件中并将其加载到 Idris 中：

```
双 : ty -> ty double=x+x
```

您会发现 Idris 拒绝此定义，并显示以下错误消息：

```
Generic.idr:2:8:
使用预期类型检查 double 的右侧时
    泰
ty 不是数字类型
```

问题在于，与身份不同， double需要了解其输入x的一些信息，特别是它是数字的。您只能对数字类型使用算术运算符，因此您需要约束ty使其仅代表数字类型。以下清单显示了如何执行此操作。

清单 2.3 一个泛型类型,受限于数字类型 (Generic.idr)

```
double : Num ty => ty -> ty double=x+x
```

←
函数类型主体部分前面的 Num ty 表示
ty 只能代表数字类型。

类型Num ty => ty -> ty可以读作“在ty是数字类型的约束下，输入类型为ty，输出类型为ty的函数”。

类型约束 泛型类型的约束可以使用接口进行用户定义，我们将在第 7 章深入介绍。这里， Num 是 Idris 提供的接口。可以为接口提供特定类型的实现，并且Num接口具有数字类型的实现。

也许令人惊讶的是，算术和比较运算符不是 Idris 中的原始运算符，而是具有受限泛型类型的函数。中缀运算符，例如+，

`==`, 和`<=`实际上是具有两个输入的函数,正如您可以通过检查它们的类型看到的那样
在REPL:

```
伊德里斯> :t (+)
(+) : Num ty => ty -> ty -> ty

伊德里斯> :t (==)
(==) : Eq ty => ty -> ty -> Bool

伊德里斯> :t (<=)
(<=) : Ord ty => ty -> ty -> Bool
```

除了数字类型的Num之外,您还可以在此处看到由

伊德里斯:

`Eq`声明类型必须支持相等和不等运算符, `==`
和`=`。
`Ord`声明该类型必须支持比较运算符`<`、`<=`、`>`和`>=`。

中缀运算符和运算符部分

Idris 中的中缀运算符不是语法的原始部分,而是由函数定义的。将运算符放在括号中,就像REPL 示例中的`(+)`、`(==)`和`(<=)`一样,意味着它们将被视为普通函数语法。例如,您可以

将`(+)`应用于一个参数:

```
伊德里斯> :t (+) 2
(+) 2 : 整数 -> 整数
```

中缀运算符也可以使用运算符部分部分应用:

`(< 3)`给出一个函数,返回其输入是否小于 3。
`(3 <)`给出一个函数,它返回 3 是否小于其输入。

因此,括号中只有一个参数的运算符被认为是一个需要另一个缺失参数的函数。

2.2.5 高阶函数类型

函数的参数或返回类型没有限制。

你已经看到了多个参数的函数是如何真正成为函数的

返回具有函数类型的东西。类似地,函数可以将函数作为参数。这样的函数称为高阶函数。

高阶函数可用于为重复的编程模式创建抽象。例如,假设您定义了一个四倍函数

它输入任何数字,使用双精度:

```
四倍 : Num a => a -> a
四倍 x = 双倍 (双倍 x)
```

或者说你有一个代表任何几何形状的Shape类型和一个函数

`rotate : Shape -> 将形状旋转 90 度的形状。你可以定义`

`将形状旋转 180 度的turn_around函数,如下所示:`

`turn_around : 形状 -> 形状`

`turn_around x = 旋转 (旋转 x)`

这些函数中的每一个都具有完全相同的模式,但它们适用于不同的输入

类型。您可以使用高阶函数来捕获此模式以应用函数

争论两次。下一个清单给出了两次函数的定义,以及

具有四倍和旋转的新定义。

清单 2.4 使用高阶函数 (HOF.idr) 定义四重和旋转

两次将一个函数作为它的第一个参数,
并将应用该函数的参数作为它的第二个参
数。

两次:
 $(a \rightarrow a) \rightarrow a \rightarrow a$
 $\text{两次 } fx = f(fx)$

形状 : 类型
旋转:形状 -> 形状

该定义遵循与 quadruple 和
turn_around 的初始定义完全相同的模
式,但具有通用函数 f。

这些是没有定义的类型声明。

四倍:Num a => a -> a
四倍 = 两倍双倍

这通过直接用 “double” 实例化 “twice” 来
实现四倍。

turn_around : 形状 -> 形状
turn_around = 两次旋转

这通过使用 “rotate” 直接实例化两次来实现
turn_around。

在第 1 章中,我介绍了“空洞”的概念,它是不完整的函数定义。清单 2.4 中没有定义的类型声明Shape和rotate是

视为孔。它们允许您尝试一个想法(例如如何实施
turn_around在旋转方面)没有完全定义类型和功能。

定义中的部分应用

在清单 2.4 中, quadruple 和 turn_around 有函数类型 Num a => a -> a 和 Shape -> Shape 分别,但在它们的定义中都没有参数。

检查定义时的唯一要求是定义的两边
必须具有相同的类型。您可以通过查看
在 REPL 中定义左右两边的类型。你有以下
定义:

`turn_around = 两次旋转`

(继续)

通过检查 REPL 中的类型,您可以看到turn_around和两次旋转都具有相同的类型:

```
伊德里斯> :t turn_around turn_around :
形状 -> 形状
Idris> :t 两次旋转
两次旋转:形状 -> 形状
```

quadruple和turn_around的定义都使用偏应用,如 2.2.2 节所述。

偏应用的另一个常见用途是为高阶函数构造参数。考虑这个例子,使用HOF.idr 并添加第 2.2.2 节中的add定义:

```
*HOF> 两次 (加 5) 10
20 : 诠释
```

这使用add函数的部分应用程序将 5 添加到Int,两次。因为两次需要一个参数的函数,而add需要两个参数,所以您可以将add应用于一个参数,以便它可以在两次应用中使用。

您还可以使用运算符部分,如第 2.2.4 节末尾所述:

```
*HOF> 两次 (5 +) 10
20 : 整数
```

请注意,在没有任何其他类型信息的情况下,Idris 默认认为Integer,如第 2.1.1 节所述。

2.2.6 匿名函数

使用高阶函数时,将匿名函数作为参数传递通常很有用。匿名函数通常是您只希望使用一次的小函数,因此无需为其创建顶级定义。

例如,您可以传递一个匿名函数,将其输入平方
两次:

```
*HOF> 两次 (\x => x * x) 2
16 : 整数
```

匿名函数以反斜杠\后跟参数列表引入。如果你检查前面的匿名函数的类型,你会看到它有一个函数类型:

```
*HOF> :t \x => x * x \x => x * x : 整数 ->
整数
```

匿名函数可以接受多个参数,并且可以选择为参数指定显式类型:

功能:Idris 程序的构建块

```
*HOF> :t \x : Int, y : Int =>x+y \x, y =>x + y : Int -> Int -> Int
```

请注意,输出未明确显示类型。

2.2.7 局部定义:let 和 where

随着函数变得越来越大,将它们分解成更小的定义通常是个好主意。Idris 为局部定义变量和函数提供了两种结构: let 和 where。

定义范围

图 2.2 说明了let 绑定的语法,它定义了局部变量。

如果您在REPL 中评估此表达式,您将看到以下内容:

```
Idris> 让 x = 50 在 x + x
100:整数
```

下一个清单显示了一个更大的let in 示例。它定义了一个更长的函数,它接受两个字符串并返回较长字符串的长度。它使用let 记录每个输入的长度。

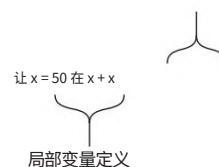


图 2.2 局部变量定义:在in关键字后面的表达式中, x的值为50。

清单 2.5 带有let 的局部变量(Let_Where.idr)

```
更长: String -> String -> Nat long word1 word2 = let len1
= length word1 len2 = length word2 in if len1 > len2 then
len1 else len2
```

记录第一个单词的长度
记录第二个字的长度 返回其中最长的一个。

多个 LETS在一个let 块中可以有多个定义。例如,在清单 2.5 中,在let 块中定义了两个局部变量。

而let 块包含局部变量定义,其中块包含局部函数定义。清单 2.6 显示了实际情况。它定义了一个函数来计算三角形斜边的长度,使用勾股定理和局部平方函数。

清单 2.6 带有where (Let_Where.idr)的局部函数定义

```
毕达哥拉斯:双 -> 双 -> 双毕达哥拉斯 xy = sqrt (平方x +平方y)
```

← sqrt 在 Prelude 中定义。

在哪里

正方形:双 -> 双正方形 x=x*x

← 这个定义只在毕达哥拉斯的范围内可见。

通常, let可用于将复杂表达式分解为更小的子表达式,而where可用于定义仅与本地上下文相关的更复杂的函数。

2.3 复合类型复合类型由其他类型组

成。在本节中,我们将了解 Idris 提供的两种最常见的复合类型:元组和列表。

2.3.1 元组

元组是一个固定大小的集合,集合中的每个值都可以有不同的类型。例如,一对Integer和String可以写成如下:

```
伊德里斯> (94, "页")
(94, Pages) : (整数, 字符串)
```

元组被写成一个括号、逗号分隔的值列表。请注意,对(94, Pages)的类型是(Integer, String)。元组类型使用与元组值相同的语法编写。

fst和snd函数分别从一对中提取第一项和第二项:

```
伊德里斯> :let mypair = (94, Pages)
```

```
伊德里斯> fst mypair
94: 整数
```

```
伊德里斯> snd mypair
"页面": 字符串
```

fst和snd都具有泛型类型,因为对可以包含任何类型。您可以在REPL 中检查每个的类型:

```
伊德里斯> :t fst
fst : (a, b) -> a
```

```
伊德里斯> :t snd
snd : (a, b) -> b
```

例如,您可以将fst的类型解读为“给定一对a和 a b,返回类型为 a 的值”。在这些类型中,您知道a和b都是变量,因为它们以小写字母开头。

元组可以有任意数量的组件,包括零:

```
Idris> ( x, 8, String) ( x, 8, String):
(Char, Integer, Type)
```

```
伊德里斯> () () :
()
```

空元组()通常称为“单元”,其类型称为“单元类型”。

请注意,语法被重载,Idris 将根据上下文决定()是指单位还是单位类型。

REPL 中的颜色您可能已经

注意到 REPL 中的某些值和类型的颜色不同,尤其是在评估空元组() 时。这是语义突出显示,它指示子表达式是类型、值、函数还是变量。默认情况下,REPL 显示如下:

类型为蓝色。 值 (更

准确地说,数据构造函数,我将在第 3 章中解释) 是红色的。 功能为绿色。 变量为洋红色。

如果这些颜色不符合您的喜好或难以区分 (例如,如果您是色盲),您可以使用:colour命令更改设置。

元组也可以任意深度嵌套:

```
Idris> (( x ,8),(String, y ,100), Hello )(( x ,8),(String, y ,100),
Hello )
: ( (字符,整数) , (类型,字符,整数) ,字符串)
```

元组和对在内部,除了空

元组之外的所有元组都存储为嵌套对。也就是说,如果你写成(1, 2, 3, 4), Idris 会以与(1, (2, (3, 4)))相同的方式处理它。

REPL 将始终以非嵌套形式显示元组:

```
伊德里斯> (1, (2, (3, 4)))
(1, 2, 3, 4) : (整数, 整数, 整数, 整数)
```

2.3.2 列表

列表和元组一样,是值的集合。与元组不同,列表可以是任意大小,但每个元素必须具有相同的类型。列表以逗号分隔的方括号中的值列表形式编写,如下所示。

```
伊德里斯> [1, 2, 3, 4]
```

```
[1, 2, 3, 4] : 列出整数
```

```
伊德里斯> [ “一”、“二”、“三”、“四” ]
```

```
[ “一”, “二”, “三”, “四” ] : 列表字符串
```

每个表达式的类型, List Integer 和 List String, 表示 Idris 为列表推断的元素类型。在类型驱动开发中,我们通常会先给出类型,然后再编写满足该类型的对应值或函数。在REPL 中,对每个值都执行此操作会很方便,因此 Idris 会尝试

推断给定值的类型。不幸的是,这并不总是可能的。例如,如果你给它一个空列表,Idris 不知道元素类型应该是什么:

```
Idris> [] (输入
    入) :无法将参数 elem 推断为 []
```

此错误消息意味着 Idris 无法计算出空列表[]的元素类型(恰好名为elem)。在这种情况下,可以通过给出显式类型来解决问题,使用:

```
Idris> (List Int) []
[]:列表整数
```

与字符串一样,列表可以与++运算符连接,前提是 oper 和 s 具有相同的元素类型:

```
伊德里斯> [1, 2, 3] ++ [4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7] :列表整数
```

您可以使用::(发音为“cons”)运算符将元素添加到列表的前面:

```
伊德里斯> 1 :: [2, 3, 4]
[1, 2, 3, 4] :列出整数
伊德里斯> 1 :: 2 :: 3 :: 4 :: []
[1, 2, 3, 4] :列出整数
```

列表的语法糖::操作符是从头元素

和尾元素构造列表的原始操作符,而Nil是空列表的原始名称。因此,列表可以采用以下两种规范形式之一:

Nil,空列表 x :: xs,其中
x是一个元素, xs是另一个列表

因为这可能会变得非常冗长,所以 Idris 为列表提供了语法糖。由方括号内的逗号分隔元素组成的列表文字被脱糖为这些原始形式。例如, []直接脱糖为Nil, [1, 2, 3]脱糖为1 :: (2 :: (3 :: Nil))。

还有一个更简洁的数字范围表示法。这里有一些例子:

[1..5]扩展到列表[1, 2, 3, 4, 5]。 [1,3..9]扩展到列表[1, 3, 5, 7, 9]。 [5,4..1]扩展到列表[5, 4, 3, 2, 1]。

更一般地, [n..m]给出n和m之间的数字列表, [n,m..p]给出n和p之间的数字列表,其步长由n和m之间的差给出。

2.3.3 带列表的函数

我们将在下一章更深入地讨论列表,包括如何在列表上定义函数,但 Prelude 中定义了几个有用的函数。让我们来看看其中的一些。

类型为String -> List String的words函数将字符串转换为字符串的空格分隔组件²:

```
Idris> 话 " Twas brillig, and the slipy toves"
[ Twas , brillig, , and , the , slithy , toves ]: 列表字符串
```

类型为List String -> String的unwords函数则相反,将单词列表转换为字符串,其中单词由空格分隔:

```
Idris> unwords [ "一"、"二"、"三"、"四!" ]
"一二三四！" : 细绳
```

您已经看到了用于计算字符串长度的长度函数。还有一个List a -> Nat类型的重载长度函数,它给出列表的长度:

```
Idris> 长度 [ "一"、"二"、"三"、"四!" ]
4: 纳特
```

您可以使用长度和单词为字符串编写字数统计函数:

```
wordCount : String -> Nat wordCount str = length
(words str)
```

map函数是一个高阶函数,它将函数应用于列表中的每个元素。它的类型为(a -> b) -> List a -> List b。此示例查找列表中每个单词的长度:

```
Idris> 地图长度 (单词 "这些单词有多长? ")
[3, 4, 3, 5, 6] : 列表 Nat
```

您可以使用map和length编写一个函数来获取字符串列表中每个元素的长度:

```
allLengths : List String -> List Nat allLengths strs = 地图长度 strs
```

地图类型

如果您检查 REPL 中的地图类型,您会看到一些稍微不同的东西:

```
Idris> :t map map : Functor
f => (a -> b) -> fa -> fb
```

²单词name 来自 Haskell 库中的类似函数。许多 Idris 函数名称如下 Haskell 术语。

(继续)

这样做的原因是map可以在各种结构上工作,而不仅仅是列表,所以它有一个受约束的泛型类型。您将在第 7 章中了解Functor ,但目前在这种类型中将f视为List是很好的。

filter函数是另一个根据布尔函数过滤列表的高阶函数。它的类型为 $(a \rightarrow \text{Bool}) \rightarrow \text{List } a \rightarrow \text{List } a$ 并返回输入列表中函数返回True 的所有内容的新列表。例如,在列表中查找所有大于 10 的数字的方法如下:

```
伊德里斯>过滤器 (> 10)[1,11,2,12,3,13,4,14]
[11, 12, 13, 14] 列出整数
```

重载函数您已经在字符串和列表

中看到了长度函数。这是可行的,因为 Idris 允许函数名称被重载以适用于多种类型。您可以通过检查REPL中的长度类型来查看发生了什么:

```
*lists> :t 长度
Prelude.List.length : 列表 a -> Nat Prelude.Strings.length :
String -> Nat
```

实际上,有两个函数叫做长度。 Prelude.List和Prelude.Strings前缀是定义这些函数的命名空间。Idris 从使用它的上下文中决定需要哪个长度函数。

类型为 $\text{Num } a \Rightarrow \text{List } a \rightarrow a$ 的sum函数计算数字列表的总和:

```
伊德里斯>总和[1..100]
5050.整数
```

sum的类型表明输入List的每个元素都必须具有相同的类型a,并且该类型受Num 约束。

您现在知道的足够多,能够定义一个名为average的函数来计算字符串中单词的平均长度。这个函数在下一个清单中定义,它显示了完整的 Idris 文件 Average.idr。

清单 2.7 计算字符串中的平均字长 (Average.idr)

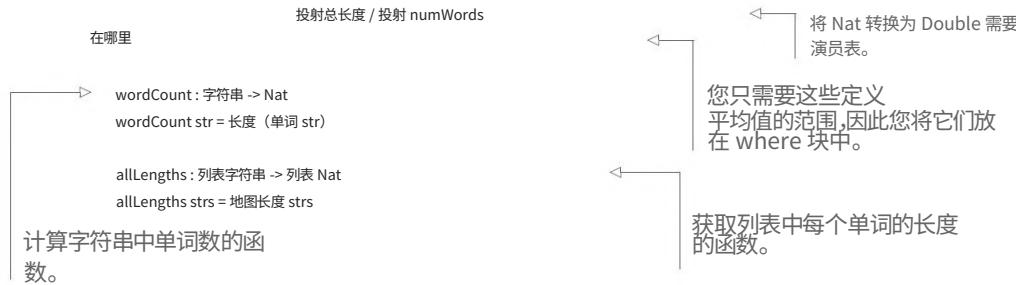
这是一个
模块声
明。

```
模块 平均
    导出平均
    值:字符串 -> 双倍平均值 str = let numWords =
        wordCount str
        totalLength = sum (allLengths (words str)) in
        totalLength / numWords
```

export 关键字表示平均的定义是
从模块中导出的。

复合类型

45

**模块和命名空间**

通过将模块声明添加到 **Average.idr** 的顶部,您声明了一个命名空间对于模块中的定义。在这里,它意味着平均函数是**Average.average**。模块声明必须是第一件事在文件中。如果没有声明,Idris 将调用模块 **Main**。

模块允许您将较大的 Idris 程序在逻辑上划分为多个源文件,每个人都有自己的目的。可以使用**import**语句导入它们。为了例子:

```

import Average 将从 Average.idr 导入定义,前提是
Average.idr 要么在当前目录中,要么在 Idris 的其他路径中
可以找到。
import Utils.Average 将在名为 Utils 的子目录中导入来自 Average.idr 的定义,前提是该文件和子目
录存在。

```

模块本身可以组合成包并单独分发。从技术上讲,**Prelude** 是在一个名为**Prelude**的模块中定义的,该模块本身导入其他几个模块,它们是名为**prelude**的包的一部分。你可以学习更多关于包以及如何从<http://idris-lang.org/documentation/packages>上的 Idris 包文档创建自己的包。

2.4 一个完整的 Idris 程序到目前为止，

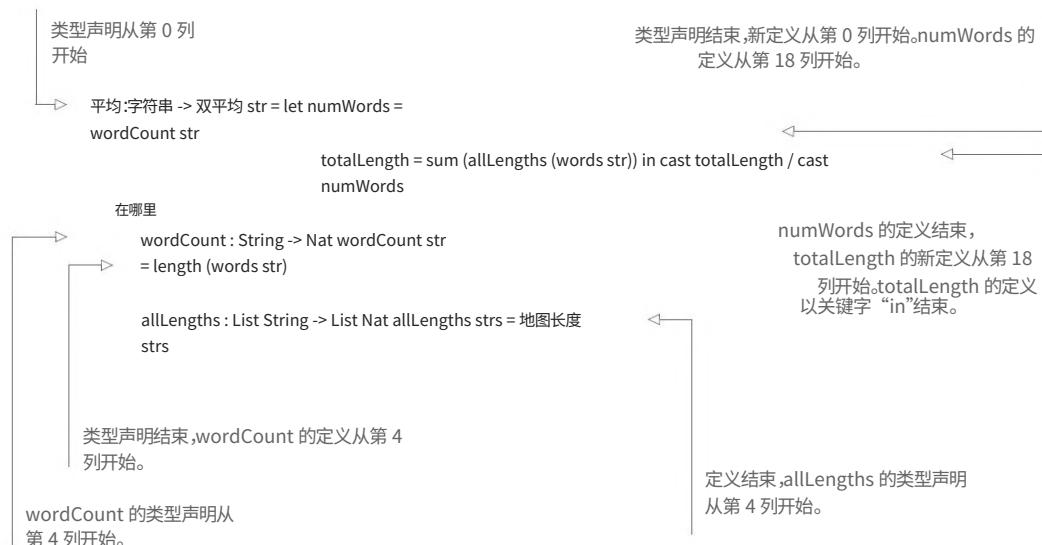
您已经了解了如何使用内置类型编写函数以及对这些类型的一些基本操作。函数是 Idris 程序的基本构建块，所以现在您已经编写了一些简单的函数，是时候看看如何将它们组合在一起构建一个完整的程序了。

2.4.1 空白意义：布局规则

空格，特别是缩进，在 Idris 程序中很重要。与其他一些语言不同，没有大括号或分号来指示表达式、类型声明和定义的开始和结束位置。相反，在任何定义和声明列表中，都必须从完全相同的列开始。清单 2.8 说明了根据该规则在包含先前平均值定义的文件中定义和声明的开始和结束位置。

空格和制表符 空格意义的一个复杂问题是制表符的大小可以在不同的编辑器中设置不同，并且 Idris 期望制表符和空格的使用保持一致。为了避免与制表符大小混淆，我强烈建议您将编辑器设置为用空格替换制表符！

清单 2.8 应用于平均值的布局规则 (Average.idr)



例如，如果 allLengths 缩进了一个额外的空格，如清单 2.9 中所示，它将被视为 wordCount 先前定义的延续，因此会无效。

一个完整的 Idris 程序

清单 2.9 布局规则,应用不正确

```
wordCount : String -> Nat wordCount str
= length (words str)

allLengths : List String -> List Nat allLengths strs = 地图长度 strs
```

allLengths 缩进了一个空格,所以这一行被认为是在 wordCount 定义的延续。

2.4.2 文档注释

与任何其他语言一样,注释定义是一种很好的形式,可以让代码读者了解函数的用途并记录它们的工作方式。Idris 提供了三种注释:

单行注释,以-- (两个减号)开头。这些评论
继续到行尾。

多行嵌套注释,以{-开头,以-}结尾。 文档注释,用于为func提供文档

REPL中的选项和类型。

前两种类型的注释是常规的,只会导致被注释的部分被忽略 (语法与 Haskell 中的注释语法相同)。

另一方面,文档注释使文档在REPL 中可用,可通过:doc命令访问。您可以查看我们目前遇到的一些类型和函数的文档。例如, :doc fst pro 产生以下输出:

```
伊德里斯> :doc fst
Prelude.Basics.fst : (a, b) -> a
    返回一对的第一个元素。
    函数是总计
```

此输出包括 fst 的完全限定名称,表明它是在模块Prelude.Basics 中定义的,并声明该函数是总的,这意味着它保证为所有输入产生结果。

您还可以获得类型的文档。例如, :doc List给出以下输出:

```
Idris> :doc List 数据类型
Prelude.List.List : Type -> Type Generic lists

构造函数:
无.列出元素
空列表
(:) : 元素 -> 列表元素 -> 列表元素
一个非空列表,由一个头元素和列表的其余部分组成。中缀 7
```

同样,这给出了Prelude.List.List 类型的完全限定名称。它还提供了构造函数,它们是构造列表的原始方式。最后,对于::运算符,它给出了固定性,表明该运算符是右结合 (中缀)并具有优先级 7。我将在第 3 章中更详细地描述运算符的优先级和结合性。

介绍了产生此文档的文档注释

带有三个竖线, |||。例如,您可以按如下方式记录平均值:

```
|||计算字符串中单词的平均长度。平均:字符串-> 双
```

然后, :doc average将产生以下输出:

```
*Average> :doc average
Average.average : (str : String) -> Double
    计算字符串中单词的平均长度。
    函数是总计
```

您可以通过给它一个名称str来引用平均参数,并在注释中使用@str 引用该名称:

```
|||计算字符串中单词的平均长度。 ||| @str 包含由空格分隔的单词的字符串。平均 : (str : String) -> Double 平均
str = let numWords = wordCount str
```

```
totalLength = sum (allLengths (words str)) in cast totalLength / cast
numWords
```

这使得:doc average产生了一些信息量更大的输出:

```
*Average> :doc average Main.average :
(str : String) -> Double 计算字符串中单词的平均长度。
```

```
论据:
str : String -- 包含由空格分隔的单词的字符串。
函数是总计
```

TOTALITY CHECKING注意:doc average报告平均值是总计。

Idris 检查每个定义的整体性。总体检查的结果对类型驱动开发有几个有趣的影响,我们将在本书中讨论,特别是在第 10 章和第 11 章。

2.4.3 互动节目

已编译 Idris 程序的入口点是main函数,定义在Main模块中。也就是说,它是具有完全限定名称Main.main的函数。它必须具有IO () 类型,这意味着它返回一个产生空元组的IO操作。

你已经看过“你好,伊德里斯世界!”程序:

```
main : IO () main =
putStrLn "Hello Idris World!"
```

这里，`putStrLn`是一个`String -> IO ()`类型的函数，它接受一个字符串作为参数并返回一个输出该字符串的`IO`动作。我们将在第 5 章深入讨论`IO`操作，但在此之前，您将能够使用`repl`函数（以及它的一些变体，如第 4 章中所见）编写完整的交互式 Idris 程序由前奏曲：

```
Idris> :doc repl
Prelude.Interactive.repl : (prompt : String) ->
    一个基本的读取-评估-打印循环
    论据：
        prompt : String -- 要显示的提示
        onInput : String -> String -- 在读取输入时运行的函数, 将字符串返回到输出
```

这允许您编写重复显示提示、读取一些输入并通过在其上运行`String -> String`类型的函数来产生一些输出的程序。例如，下一个清单是一个重复读取字符串然后反向打印字符串的程序。

清单 2.10 以交互方式反转字符串 (Reverse.idr)

模块主要

主:IO ()主=复制> “

撤销

你可以使用`:exec`命令在REPL编译和运行这个程序。请注意，程序将无限循环，但您可以通过使用`Ctrl-C`中断程序来退出：

```
*反向>.执行
> 你好!
!olleh>再见eybdoog>
```

在本章结束时，您将编写一个导入`Average`模块的程序，从控制台读取一个字符串，并显示字符串中每个单词的平均字母数。一个难点是`average`函数返回一个`Double`，但是`repl`需要一个`String -> String`类型的函数，所以不能直接使用`average`。

但是，通常可以使用`show`函数将值转换为字符串。让我们使用`:doc`来看看它：

```
Idris> :doc show
Prelude.Show.show : Show ty => (x : ty) -> String 将值转换为其字符串表示形式。
```

请注意，这是一个受约束的泛型类型，这意味着类型`ty`必须支持`Show`接口，这对 Prelude 中的所有类型都是如此。

使用它,您可以编写一个showAverage函数,该函数使用average来获取平均字长,并将其显示在格式良好的字符串中。完整的程序是在下面的清单中给出。

清单 2.11 以交互方式显示平均字长 (AveMain.idr)

模块主要

从 Average.idr 导入定义

进口平均

showAverage : 字符串 -> 字符串

showAverage str = 平均字长为: show (average str) ++ \n

++ 计算要输出的字符串的函数

一些输入。它使用 show 来转换结果
平均到一个字符串。

提示
显示

主要:我 ()
main = repl 输入一个字符串:

显示平均

← 调用计算输出的函数

同样,您可以使用:exec在REPL 上编译和运行它,然后尝试一些输入:

```
*AveMain> :exec
输入一个字符串:敏捷的棕色狐狸跳过了懒惰的狗
平均字长为 4
输入字符串:敏捷的棕狐跳过懒惰的青蛙
平均字长为 4.11111
输入一个字符串:
```

练习

1以下值的类型是什么?

```
( "A" , "B" , "C" )
[ A , B , C ]
(( A , B ), C )
```

您可以在REPL使用:t检查您的答案,但请尝试自己解决第一的。

2编写一个String -> Bool类型的回文函数,它返回是否

输入向后读取与向前读取相同。

提示:您可能会发现函数reverse : String -> String很有用。

您可以在REPL上测试您的答案,如下所示:

```
*ex_2> 回文 "赛车"
真:布尔
```

```
*ex_2> 回文 "赛车"
假:布尔值
```

3修改回文函数,使其不区分大小写。

提示:您可能会发现`toLower : String -> String`很有用。

您可以在REPL上测试您的答案,如下所示:

```
*ex_2> 回文 “赛车”
真:布尔
```

4修改回文函数,使其仅对长度超过的字符串返回True

10 个字符。

您可以在REPL上测试您的答案,如下所示:

```
*ex_2> 回文 “赛车”
假:布尔值
```

```
*ex_2>回文 “我能看到厄尔巴岛之前”
真:布尔
```

5修改回文函数,使其仅对长度超过的字符串返回True

作为参数给出的一些长度。

提示:你的新函数的类型应该是`Nat -> String -> Bool`。

您可以在REPL上测试您的答案,如下所示:

```
*ex_2> 回文 10 “赛车”
假:布尔值
```

```
*ex_2>回文5 “赛车”
真:布尔
```

6编写一个`String -> (Nat, Nat)`类型的计数函数,它返回一对

输入中的单词数和输入中的字符数。

您可以在REPL上测试您的答案,如下所示:

```
*ex_2> 计数 “你好,伊德里斯世界！”
(3, 19) : (晚上, 晚上)
```

7编写一个类型为`Ord a => List a -> List a`的`top_ten`函数,它返回列表中最大的 10 个值。您可能会发现以

下 Prelude 功能很有用:

```
采取:Nat -> List a -> List a
排序:Ord a => List a -> List a
```

如果需要,请使用`:doc`获取有关这些功能的更多信息。

您可以在REPL上测试您的答案,如下所示:

```
*ex_2> 前十 [1..100]
[100, 99, 98, 97, 96, 95, 94, 93, 92, 91] : 列表整数
```

8编写一个`Nat -> List String -> Nat`类型的`over_length`函数,它返回
列表中长于给定字符数的字符串数。

您可以在REPL上测试您的答案,如下所示:

```
*ex_2> over_length 3 [ “一”、 “二”、 “三”、 “四” ]
2: 整数
```

9对于每一个回文数和计数,编写一个完整的程序,提示输入一个
输入,调用函数,并打印其输出。

您可以在REPL中使用:exec测试您的答案:

```
*ex_2_palindrome> :exec
输入一个字符串:在我看到厄尔巴之前我能做到吗
真的
输入一个字符串:女士,我是亚当
错误的
输入一个字符串:
```

2.5 总结

Prelude 定义了许多基本类型和功能,并由所有 Idris 程序自动导入。 Idris 提供基本的数值类型Int、 Integer、 Nat和Double,以及布尔类型Bool、字符类型Char和字符串类型String。 可以使用cast函数在兼容类型之间转换值,并且

可以使用函数给定显式类型。 元组是固定大小的集合,其中每个元素
可以是不同的类型。 列表是可变大小的集合,其中每个元素都具有相同的类型。 函数类型有一种或多种输入类型
和一种输出类型。 函数类型可以是泛型的,这意味着它们可以包含变量。可以限制这些变量以允许更小的类型集。
高阶函数是其中一个参数本身是

功能。

函数由必需的类型声明和定义组成。函数定义是定义要在评估期间使用的重写规则的方程。 空白在 Idris 程序中很重要。
块中的每个定义必须从完全相同的列开始。

可以在REPL中使用:doc命令访问函数文档。

Idris 程序可以被分成独立的源文件,称为模块。 Idris 程序的入口点是main函数,它的类型必须为
IO(),并在模块Main 中定义。可以通过应用main中的repl函数来编写简单的交互式程序。

第2部分

核心伊德里斯

现在你已经有了一些在 Idris 编写程序的经验,是时候开始深入探索类型驱动的开发。在这一部分中,您将了解 Idris 的核心特性,并在开发过程中获得一些经验类型驱动开发。而不是向您展示正确的完整程序从一开始,我将通过以下过程向您展示如何以交互方式构建程序类型,定义,细化:

类型 为函数编写类型。 定义 为函数

创建一个初始定义,可能包含
孔。

细化 通过填充孔完成定义,可能修改

随着您对问题的理解的发展,键入。

在第 3 章中,您将学习交互式开发的基础知识;然后,在第 4 章,您将学习定义自己的数据类型并构建更大的程序在他们旁边。第 5 章展示了如何编写与外部世界,使用类型将评估与执行分开。后面的章节介绍了类型驱动开发中更高级的概念,包括

第 6 章中的类型级计算,使用受约束的泛型类型
第 7 章,在第 8 章和第 9 章中描述和证明程序的属性,
并在第 10 章中使用视图定义数据结构的替代遍历。

在第 2 部分结束时,您将了解

伊德里斯。

Machine Translated by Google

与类型的交互式开发

本章涵盖

通过模式匹配定义函数 Atom 中类型驱动的
交互式编辑增加函数类型的精度使用向量进行
实际编程

您现在已经了解了如何定义简单的函数,以及如何将它们构建成完整的程序。在本章中,我们将开始对类型驱动开发进行更深入的探索。首先,我们将了解如何使用 Prelude 中的现有类型 (例如列表) 编写更复杂的函数。然后,我们将研究使用 Idris 类型系统为函数提供更精确的类型。

在类型驱动开发中,我们遵循“类型、定义、改进”的过程。当你第一次编写类型时,你会在本章中看到这个过程在起作用,并且尽可能地总是有一个类型正确的函数定义,如果可能是不完整的,并逐步改进它直到它完成。每个步骤将被广泛描述为以下三个之一:

类型 要么写一个类型来开始这个过程,要么检查一个洞的类型来决定如何继续这个过程。

定义 通过创建定义的轮廓或将其分解为更小的组件来创建函数定义的结构。

细化 通过填充一个洞或制作它来改进现有的定义
打字更精确。

在本章中,我将介绍 Atom 文本编辑器中的交互式开发,它协助这个过程。Atom 提供了一种交互式编辑模式,可以与正在运行的 Idris 系统进行通信,并使用类型来帮助指导功能开发。Atom 还提供了一些结构编辑功能和上下文信息带有孔的函数,并且当类型足够精确时,甚至可以完成大部分功能为您服务。因此,我将通过介绍 Atom 中的交互式编辑模式来开始本章。

编辑器模式 虽然我们将使用 Atom 来编辑 Idris 程序,但我们将使用的交互功能由 Idris 本身提供。Atom 集成有效通过与后台运行的 Idris 进程通信。此过程作为编辑器的子进程运行,因此它独立于您可能使用的任何REPL 有跑步。因此,将 Idris 支持添加到其他文本编辑器和类似的编辑模式目前存在于 Emacs 和维姆。在本书中,为了保持一致性,我们将坚持使用 Atom,但每个命令都直接映射到其他编辑器中的相应命令。

3.1 Atom 中的交互式编辑

您在第 1 章中看到 Idris 程序可以包含孔,这些孔代表尚未编写的函数定义。这是程序的一种方式可以交互开发:您编写一个包含孔洞的不完整定义,检查孔的类型以查看 Idris 对每个孔的期望,然后继续用更多代码填补漏洞。伊德里斯还有其他几种方法可以通过将交互式开发功能与文本编辑器集成来帮助您:

添加定义 给定一个类型声明,Idris 可以添加一个框架定义
满足该类型的函数。

案例分析 给定一个带参数的骨架函数定义,Idris 可以使用
这些参数的类型以帮助通过模式匹配定义函数。

表达式搜索 给定一个类型足够精确的孔,Idris 可以尝试查找
满足孔类型的表达式,细化定义。

在本节中,我们将开始编写一些更复杂的 Idris 函数,使用其交互式编辑功能以类型导向的方式逐步开发这些函数。

我们将使用 Atom 文本编辑器,因为有一个可用于编辑 Idris 的扩展程序,可以直接从默认的 Atom 发行版安装。其余的部分本章假设您已经启动并运行了交互式 Idris 模式。如果不是,请按照附录 A 中的说明安装 Atom 和 Idris 模式。

3.1.1 交互命令总结

Atom 中的交互式编辑涉及编辑器中的许多键盘命令，总结在表 3.1 中。

命令助记符对于每个命令，Atom 中的快捷方式是按下 **Ctrl+Alt** 和命令的第一个字母。

表 3.1 Atom 中的交互式编辑命令

捷径	命令	描述
Ctrl+Alt+A	添加定义	为光标下的名称添加骨架定义
Ctrl+Alt+C	案例拆分	将定义拆分为光标下名称的模式匹配子句
Ctrl+Alt+D	文档	显示光标下名称的文档
Ctrl+Alt+L	提升孔	将一个洞提升到顶层作为一个新的函数声明
Ctrl+Alt+M	匹配	用匹配中间结果的 case 表达式替换空洞
Ctrl+Alt+R	重新加载	重新加载和类型检查当前缓冲区
Ctrl+Alt+S	搜索	搜索满足光标下孔名称类型的表达式
Ctrl+Alt+T	类型检查名称	显示光标下名称的类型

3.1.2 通过模式匹配定义函数

到目前为止，我们看到的所有函数定义都涉及一个方程
定义该函数的行为。例如，在上一章你写了一个
计算列表中每个单词的长度的函数：

```
allLengths : 列表字符串 -> 列表 Nat
allLengths strs = 地图长度 strs
```

在这里，您使用了 Prelude 中定义的函数（地图和长度）来检查列表。
但是，在某个阶段，您将需要一种更直接的方法来检查值。后
所有，像 map 和 length 这样的函数需要以某种方式定义自己！

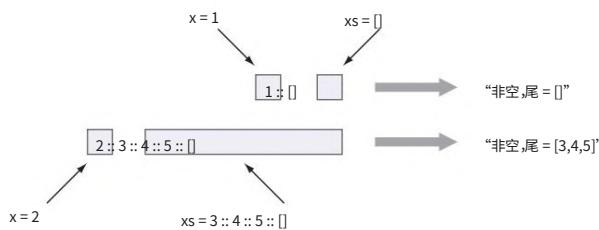
通常，您通过对可能的值进行模式匹配来定义函数
函数的输入。例如，您可以定义一个函数来反转 Bool，如下所示：

```
反转:布尔 -> 布尔
反转 False = True
反转真 = 假
```

Bool 类型的可能输入是 True 和 False，所以在这里你实现 invert
通过列出可能的输入并给出相应的输出。图案还可以
包含变量，如下面的函数所示，如果

给定一个空列表或“非空”，如果给定一个非空列表，则后跟尾部的值：

图 3.1 说明了在 `describeList` 中模式如何匹配输入 `[1]`（它是 `1 :: []` 的语法糖）和 `[2,3,4,5]`（它是 `2 :: 3 :: 4 :: 5 :: []` 的语法糖）。



列表的命名约定按照惯例，在使用列表和类似列表的结构时，Idris 程序员使用以“s”结尾的名称（表示复数），然后使用单数来指代各个元素。因此，如果您有一个名为 `things` 的列表，您可以将列表中的一个元素称为 `thing`。

函数定义由一个或多个与该函数的可能输入相匹配的方程组成。如果您通过直接检查列表而不是使用 `map` 来实现 `allLengths`，您可以看到这对列表是如何工作的。

清单 3.1 通过列表上的模式匹配计算字长 (WordLength.idr)

如果输入列表为空，则输出列表也将为空。

如果输入列表有一个头元素 `x` 和一个尾元素 `xs`，则输出列表将以 `x` 的长度作为头，然后递归地将 `xs` 的长度列表作为尾。

要详细了解如何构建此定义，您将在 Atom 中以交互方式构建它。

每个步骤都可以概括为 Type（创建或检查类型）、Define（定义或将其分解为单独的子句）或 Refine（通过填充孔或使其类型更精确来改进定义）。

1类型 首先，启动 Atom 并创建一个 `WordLength.idr` 文件，其中包含 `allLengths` 的类型声明。那只是以下几点：

您还应该在单独的终端中启动 Idris REPL ,以便您可以
类型检查并测试您的定义：

```
$ idris WordLength.idr
--_
/_/_/_/(_)
////_/_/_/_/_/_/(_)/_/
\_,/_/_/_/_/
1.0 版
http://www.idris-lang.org/
类型 : ?求助

Idris 是完全没有保证的免费软件。
有关详细信息,请键入:保修。
孔:Main.allLengths
*字长>
```

2 定义 - 在 Atom 中,将光标移到名称 allLengths 上,然后按 Ctrl-Alt-A。

这将添加一个骨架定义,您的编辑器缓冲区现在应该包含
以下:

```
allLengths : 列表字符串 -> 列表 Nat
allLengths xs = ?allLengths_rhs
```

骨架定义始终是一个子句,其左侧列出了适当数量的参数,右侧有一个孔。伊德里斯使用

=,

各种启发式方法来选择参数的初始名称。按照惯例,
Idris 为列表选择默认名称 xs、 ys 或 zs 。

3 Type - 您可以通过按 Ctrl-Alt-T 来检查 Atom 中的孔类型

将光标悬停在要检查的孔上。如果您检查的类型

allLengths_rhs 洞,你应该看到这个:

```
xs : 列表字符串
-----
allLengths_rhs : 列表 Nat
```

4 定义 您将通过检查名为 xs 的列表参数来编写此定义,

直接地。这意味着对于列表可以采用的每种形式,您需要解释

当它是这种形式时如何测量字长。告诉编辑你

要检查第一个参数,请在 Atom 中按 Ctrl-Alt-C 并将光标悬停在

第一个参数位置的变量 xs 。这将定义扩展为

给出参数 xs 可以采用的两种形式:

```
allLengths : 列表字符串 -> 列表 Nat
allLengths [] = ?allLengths_rhs_1
allLengths (x :: xs) = ?allLengths_rhs_2
```

这是列表的两种规范形式。也就是说,每个列表必须在其中之一
这两种形式:它可以是空的(以[]形式),也可以是非空的,
包含一个头元素和列表的其余部分(格式为(x :: xs))。它是
此时将 x 和 xs 重命名为更有意义的名称是个好主意
比这些默认名称:

```
allLengths : 列表字符串 -> 列表 Nat
allLengths [] = ?allLengths_rhs_1
allLengths (word :: words) = ?allLengths_rhs_2
```

在每种情况下,右侧都有一个新孔要填充。您可以检查这些孔的类型;类型检查给出预期的返回类型和类型任何局部变量。例如,如果您检查allLengths_rhs_2 的类型,您将看到局部变量word和words 的类型,以及预期的返回类型:

```
字 :字符串
words :列表字符串
-----
allLengths_rhs_2 :列表 Nat
```

5 精炼 伊德里斯现在已经告诉您需要哪些模式。你的工作是通过填充右侧的孔来完成定义。在这种情

况下

输入是空列表,输出也是空列表,因为没有

测量长度的单词:

```
所有长度 [] = []
```

6 Refine 在输入非空的情况下,有一个单词作为第一个元素(单词),然后是列表的其余部分(单

词)。您需要返回一个

以单词的长度为第一个元素的列表。目前,您可以添加一个
列表其余部分的新孔(?rest) :

```
allLengths :列表字符串 -> 列表 Nat
所有长度 [] = []
allLengths (word :: words) = 长度 word :: ?rest
```

你甚至可以在REPL 上测试这个不完整的定义。请注意, REPL

不会自动重新加载文件,因为它独立于 Atom 中的交互式编辑运行,因此您需要使用:r命令显式重
新加载:

```
*字长>:r
类型检查 ./WordLength.idr
孔 :Main.rest
*WordLength> allLengths [ Hello , Interactive , Editors ]
5 :: ?rest :列表 Nat
```

对于孔休息,您需要计算单词的长度。你

可以通过递归调用allLengths 来完成定义:

```
allLengths :列表字符串 -> 列表 Nat
所有长度 [] = []
allLengths (word :: words) = 长度 word :: allLengths 单词
```

您现在有了一个完整的定义,您可以在重新加载后在REPL上进行测试:

```
*字长>:r
类型检查 ./WordLength.idr
*WordLength> allLengths [ Hello , Interactive , Editors ]
[5, 11, 7] :列表 Nat
```

检查 Idris 是否相信定义是完整的也是一个好主意。

```
*WordLength> :total allLengths
Main.allLengths 是总计
```

整体检查

当 Idris 成功地对一个函数进行类型检查后,它还会检查它是否认为功能是完全的。如果一个函数是总的,它保证产生一个在有限时间内,任何类型良好的输入的结果。由于停机问题,我们在第 1 章中讨论过,Idris 通常不能确定一个函数是否是全函数,但是通过分析函数的语法,它可以确定一个函数在许多特定情况下是完全的。

我们将在第 10 章和第 11 章中更详细地讨论整体性检查。
时刻,只要知道 Idris 将考虑一个函数总计,如果

它的子句涵盖了所有可能的类型良好的输入
所有递归调用都收敛于一个基本情况

正如你将在第 11 章中看到的那样,整体性的定义还允许无限期运行的交互式程序,例如服务器和交互式循环,只要他们继续在有限的时间内产生中间结果。

Idris 认为 allLengths 是总的,因为所有可能的类型良好的输入都有子句,并且对 allLengths 的递归调用的参数更小(即,比输入更接近基本情况)。

3.1.3 数据类型和模式

当您在 Atom 中按下 Ctrl-Alt-C 并将光标悬停在 a 左侧的变量上时
定义时,它对该变量执行大小写拆分,给出变量可以匹配的可能模式。但是这些模式从何而来?

每种数据类型都有一个或多个构造函数,它们是在该数据类型中构建值并给出可以与该数据匹配的模式的原始方法
类型。对于 List,有两个:

Nil,构造一个空列表
::,一个中缀运算符,从头元素和尾元素构造列表

此外,正如您在第 2 章中看到的,列表的语法糖允许列表
写为方括号中的逗号分隔值列表。因此, Nil 也可以
写成[]。

对于任何数据类型,您都可以找到构造函数,从而找到要匹配的模式
在 REPL 提示符下使用 :doc :

```
伊德里斯> :doc 列表
数据类型 Prelude.List.List : (elem : Type) -> Type
    通用列表

构造函数:
    无:列出元素
    空列表
        (::) : (x : elem) -> (xs : 列表元素) -> 列表元素
```

一个非空列表,由一个 head 元素和其余元素组成
名单。
中缀 7

ATOM中的文档您可以直接在 Atom 中获取文档
通过按 Ctrl-Alt-D,将光标放在您想要的名称上
文档。

例如,对于Bool, :doc表明构造函数是False和True:

```
伊德里斯> :doc 布尔值
数据类型 Prelude.Bool.Bool : 类型
布尔数据类型

构造函数：
假:布尔值
真:布尔
```

因此,如果您编写一个将Bool作为输入的函数,您可以提供
输入False和True 的显式案例。

例如,要编写异或运算符,您可以按照以下步骤操作: 1类型 首先给出一个类型:

xor:布尔 -> 布尔 -> 布尔

2定义 - 按 Ctrl-Alt-A 并将光标悬停在xor上以添加骨架定义:

```
xor:布尔 -> 布尔 -> 布尔
xor xy=?xor_rhs
```

3定义在 x 上按 Ctrl-Alt-C以给出x的两种可能情况:

```
xor:布尔 -> 布尔 -> 布尔
xor False y=?xor_rhs_1
xor True y=?xor_rhs_2
```

4细化 - 通过填写右侧完成定义:

```
xor:布尔 -> 布尔 -> 布尔
xor Falsey=y
xor Truey=not y
```

ATOM中的类型检查 在开发功能时,尤其是在
手写从句而不是使用交互式编辑功能,它可以
最好对到目前为止的内容进行类型检查。 Ctrl-Alt-R 命令
使用正在运行的 Idris 进程重新检查当前缓冲区。如果加载成功,Atom 状态栏会报告“文件
加载成功”。

表示无界无符号整数的Nat类型也由原始构造函数定义。在 Idris 中,自然数被定义为 0 或 1

大于 (即,后继)另一个自然数。

```
伊德里斯> :doc Nat
数据类型 Prelude.Nat.Nat : 类型
自然数:无界的无符号整数,可以是
模式匹配。
```

构造函数：
来自 纳特
零

S: 纳特 -> 纳特
接班人

数据类型和构造函数 数据类型是根据它们的构造函数定义的,你将在第 4 章详细了解。**数据类型的构造函数**

是构建该数据类型的原始方法,因此对于Nat,每个 Nat类型的值必须为零或另一个Nat 的后继。这 例如,数字3 以原始形式写为S (S Z)。也就是说,它 2的后继者(S) (写为S (S Z))。

因此,如果您编写一个将Nat作为输入的函数,您可以提供显式 对于数字零(Z)或大于零的数字 (S k,其中k是任何非负数)的情况。例如,要编写一个isEven函数,如果输入的自然数可被 2 整除,则返回 True,否则返回False ,您可以定义它

递归 (如果效率低下)如下： **1**类型 - 从给出类型

开始：

```
isEven : Nat -> Bool
```

2定义 - 按 Ctrl-Alt-A 添加骨架定义：

```
isEven : Nat -> Bool
isEven k = ?isEven_rhs
```

命名约定作为命名约定,Idris默认选择k Nat类型的变量。命名约定可由程序员设置 在定义数据类型时,你将在第 4 章看到如何做到这一点。在任何 在这种情况下,将这些变量重命名为更多名称通常是个好主意 内容丰富。

3定义 - 在 k 上按 Ctrl-Alt-C以给出k的两种可能情况：

```
isEven : Nat -> Bool
isEven Z = ?isEven_rhs_1
isEven (S k) = ?isEven_rhs_2
```

要完成定义,您必须解释输入时返回的内容 为零(Z)或当输入为非零时 (如果输入采用S k 的形式,则k 是一个变量,代表比输入小一的数字)。

4细化 - 通过填写右侧完成定义：

```
isEven : Nat -> Bool
isEven Z = 真
isEven (S k) = 不是 (isEven k)
```

您已经递归地定义了它。零是偶数,所以你返回True 输入Z。如果一个数字是偶数,它的后继是奇数,反之亦然,所以你 为输入S k返回not (isEven k)。

相互定义的函数Idris 从上到下

处理输入文件,需要在使用前定义类型和函数。这是必要的,因为依赖类型会出现复杂情况,其中函数的定义会影响类型。

然而,有时根据每个函数定义两个或多个函数是有用的
其他。这可以在相互块中实现。例如,您可以根据isOdd函数定义isEven ,反之亦然:

```
相互的
isEven : Nat -> Bool
isEven Z = 真
isEven (S k) = isOdd k

isOdd : Nat -> Bool
isOdd Z = 假
isOdd (S k) = isEven k
```

3.2 为类型增加精度:使用向量在第 1 章中,我们讨论了如何将类型作为一流的语言

结构允许我们定义更精确的类型。作为一个例子,您看到了如何通过在其类型中包含列表中元素的数量以及元素的类型来为列表提供更精确的类型。在 Idris 中,在其类型中包含元素的数量和类型的列表称为向量,定义为数据类型Vect。以下清单显示了一些示例向量。

清单 3.2 向量:长度编码为类型 (Vectors.idr) 的列表

```
导入数据.Vect

FourInts : Vect 4 Int FourInts = [0, 1, 2,
3]

SixInts : Vect 6 Int SixInts = [4, 5, 6, 7,
8, 9]

tenInts : Vect 10 Int
tenInts = 四整数 ++ 六整数
```

使用 ++ 附加向量也会在结果类型中添加它们的长度。

Vect未在 Prelude 中定义,但可以通过导入Data.Vect库模块使其可用。模块通过源文件顶部的import语句导入:

```
导入数据.Vect
```

包:前奏和基础Idris 模块可以组合成包,从中可以导入单个模块。 Prelude 定义在一个名为 prelude 的包中,所有模块都从该包中自动导入。

Idris 程序还可以访问一个名为base 的包,该包定义

为类型增加精度：使用向量

几种常用的数据结构和算法，包括Vect，但必须从其中显式导入模块。与 Idris 一起分发的软件包的最新文档可在www.idris-lang.org/documentation 获得。

3.2.1 细化 allLengths 的类型

要了解Vect的工作原理，您可以从第 3.1.2 节中细化allLengths函数的类型，以使用Vect而不是List，然后重新定义该函数。

为此，请创建一个名为 WordLength_vec.idr 的新源文件，其中仅包含 import Data.Vect 行，并将其加载到REPL 中。然后您可以查看Vect 的文档：

```
*WordLength_vec> :doc Vect 数据类型
Data.Vect.Vect : Nat -> Type -> Type
    向量：类型中具有显式长度的通用列表

构造函数：
    空向量
        无.Vect 0 a
    空向量

    (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
        一个长度为 S k 的非空向量，由一个头元素和长度为 k 的列表的其余部分组成。

    中缀 7
```

请注意，它具有与List相同的构造函数，但它们具有不同的类型，它们给出了明确的长度。长度以Nat 形式给出，因为它们不能为负数：

Nil的类型明确指出长度为Z（显示为数字点亮
此处为 0）。
::的类型明确声明长度为S k，给定一个元素和一个尾部
长度为k。

与List相同的语法糖适用于将括号中的值列表（例如[1, 2, 3]）转换为::和Nil 的序列：1 :: 2 :: 3 :: Nil，在这种情况下。事实上，这种语法糖适用于任何具有称为Nil和:: 的构造函数的数据类型。

重载名称List和Vect的

构造函数名称相同，Nil 和 :: 名称可以重载，前提是在不同的命名空间中定义具有相同名称的不同事物，这在实践中通常意味着不同的模块。Idris 将从使用名称的上下文中推断出适当的名称空间。

您可以使用以下命令明确指示需要List或Vect中的哪一个：

```
伊德里斯> (List_) [ Hello , There ]
[ Hello , There ]: 列表字符串

伊德里斯> (Vect _ _ ) [ “你好” ]
[ Hello , There ]: Vect 2 字符串
```

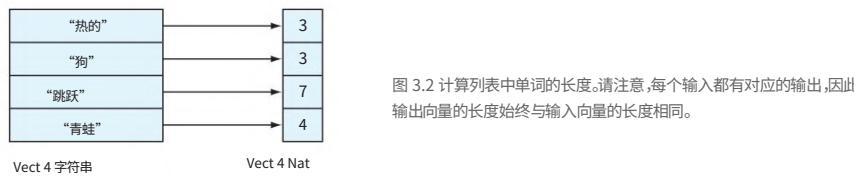
(继续)
上述表达式中的下划线 (_) 向 Idris 表明您想要
它来推断该参数的值。您可以在表达式中使用
只有一个代表该表达式的有效值。你会看到更多关于这个
在第 3.4 节中。

要使用Vect定义allLengths，您可以遵循与使用List 定义时类似的过程。不同之处在于您必须考虑

输入和输出是相关的。

图 3.2 显示了如何在输出列表中始终有一个对应于的条目输入列表中的条目。因此，您可以明确表示输出向量与输入向量的长度相同：

allLengths : Vect len 字符串 -> Vect len Nat



输入中出现的len是类型级别的变量,代表长度的输入。因为输出在类型级别使用相同的变量,所以它在输出与输入长度相同的类型。这是你可以写的方法功能:

1类型创建一个包含以下内容的 Atom 缓冲区以导入Data.Vector，并给出allLengths 的类型：

导入数据.Vect

`allLengths : Vect len 字符串 -> Vect len Nat`

2 定义和以前一样,使用 Ctrl-Alt-A 创建骨架定义:

allLengths : Vect len 字符串 -> Vect len N
allLengths xs = ?allLengths rhs

3 定义 和以前一样,在 xs 上按 Ctrl-Alt-C 告诉 Idris 你想通过 xs 上的模式匹配来定义函数:

```
allLengths : Vect len 字符串 -> Vect len Nat  
allLengths [] = ?allLengths_rhs_1  
allLengths (x :: xs) = ?allLengths_rhs_2
```

和以前一样,此时最好将变量`x`和`xs`重命名为更有意义的事情:

all_lengths : Vect len 字符串 \rightarrow Vect len Nat

```
allLengths [] = ?allLengths_rhs_1 allLengths (word :: words)
= ?allLengths_rhs_2
```

- 4**类型 - 如果您现在检查孔allLengths_rhs_1和allLengths_rhs_2的类型,您将看到比列表版本更多的信息,因为类型更精确。例如,在allLengths_rhs_1中,您可以看到唯一有效的结果是具有零元素的向量:

```
-----  
allLengths_rhs_1:Vect 0 Nat
```

在allLengths_rhs_2中,您可以看到模式变量和输出的长度是如何相互关联的,给定一个自然数n:

```
单词:字符串 k:Nat  
words : Vect k 字符串  
-----  
allLengths_rhs_2 : Vect (S k) Nat
```

也就是说,在模式 (word :: words)中, word是一个String, words是kStrings的向量,对于输出你需要提供一个长度为1+ k的Nat向量,表示为S k。

- 5**细化对于allLengths_rhs_1,只有一个长度为零的向量,即空向量,因此只有一个值可用于通过填充孔来细化定义:

```
allLengths : Vect len String -> Vect len Nat allLengths [] = [] allLengths (word :: words) = ?allLengths_rhs_2
```

- 6 Refine** - 对于allLengths_rhs_2,所需的类型是Vect (S k) Nat,因此结果必须是非空向量(使用::)。此外,您可以通过递归调用allLengths来生成Vect k Nat类型的值。您可以为结果列表中的第一个元素留一个洞,手动细化定义如下:

```
allLengths : Vect len String -> Vect len Nat allLengths [] = [] allLengths (word :: words) = ?wordlen :: allLengths words
```

这个结果中还有一个漏洞, ?wordlen,它将是第一个单词的长度:

```
单词:字符串 k:Nat 单词:  
Vect k 字符串  
-----  
wordlen : 纳特
```

- 7 Refine** - 要完成定义,请通过计算单词x的长度:

```
allLengths : Vect len String -> Vect len Nat allLengths [] = [] allLengths (word :: words) = length word :: allLengths words
```

更精确的类型描述了输入和输出的长度如何相关,这意味着交互式编辑模式可以告诉您更多有关您正在寻找的表达式的信息。您还可以通过排除任何不通过类型检查保留长度的程序来更加确信程序的行为符合预期。

NAT 和数据结构您可能会注意到Vect的构造函数和Nat 的构造函数之间的直接对应关系。当您使用::将元素添加到Vect时,您会在其长度上添加一个S构造函数。在实践中,像这样捕获数据结构的大小是Nat 的一个非常常见的用途。

为了说明更精确的类型如何排除一些不正确的程序,请考虑以下allLengths 的实现,使用List而不是Vect:

```
allLengths : 列表字符串 -> 列表 Nat allLengths xs = []
```

这是很好的类型并且 Idris 会接受它,但它不会按预期工作,因为不能保证输出列表具有与输入中的每个条目相对应的条目。另一方面,以下具有更精确类型的程序类型不好,不会被 Idris 接受:

```
allLengths : Vect n 字符串 -> Vect n Nat allLengths xs = []
```

这会导致以下类型错误,即在需要长度为n的向量时给出了一个空向量:

```
WordLength_vec.idr:4:14: 检查 allLengths 的右侧时: 类型不匹配
```

```
    Vect 0 Nat ([] 类型)  
和  
    Vect n Nat (预期类型)
```

与之前基于列表的allLengths 版本一样,您可以在REPL中检查您的新定义是否为总定义:

```
*WordLength_vec> :total allLengths Main.allLengths 是 Total
```

例如,如果您删除空列表的大小写,则您有一个类型良好但不完整的定义:

```
allLengths : Vect len String -> Vect len Nat allLengths (word :: words) = length  
word :: allLengths words
```

当你检查这个整体时,你会看到这个:

```
*WordLength_vec> :total allLengths Main.allLengths 不完整,因  
为缺少案例
```

为类型增加精度：使用向量

Totality annotations为了

增加对函数正确性的信心，您可以在源代码中注释函数必须是全部的。例如，你可以这样写：

```
总 allLengths : Vect len String -> Vect len Nat
allLengths [] = []
allLengths (word :: words)
= length word :: allLengths words
```

类型声明前的total关键字意味着如果定义不是total，Idris会报错。例如，如果您删除allLengths []案例，Idris 将报告以下内容：

```
WordLength_vec.idr:5:1:Main.allLengths
不完整,因为缺少案例
```

3.2.2 类型导向搜索：自动精炼

在上一节的第 3 步之后，您就拥有了allLengths的图案和右侧的孔洞，这是您通过直接精炼提供的：

```
allLengths : Vect n 字符串 -> Vect n Nat
allLengths [] = ?allLengths_rhs_1
allLengths (word :: words) = ?allLengths_rhs_2
```

再看一下allLengths_rhs_1和allLengths_rhs_2孔的类型和局部变量：

```
allLengths_rhs_1:Vect 0 Nat
```

```
单词:字符串 k:Nat 单词:
Vect k 字符串
```

```
allLengths_rhs_2 : Vect (S k) Nat
```

通过仔细查看类型，您可以了解如何构造值来填补这些漏洞。但不仅您在这里有更多信息，伊德里斯也有！

给定类型中的足够信息，Idris 可以搜索满足该类型的有效表达式。在 Atom 中，在allLengths_rhs_1 孔上按 Ctrl-Alt-S，您应该会看到定义已更改：

```
allLengths : Vect n String -> Vect n Nat
allLengths [] = []
allLengths (word :: words) = ?allLengths_rhs_2
```

因为长度为 0 的向量只有一个可能的值，所以 Idris 自动改进了它。

您还可以尝试对allLengths_rhs_2孔进行表达式搜索。按 Ctrl Alt-S 将光标放在allLengths_rhs_2 上，您应该会看到：

```
allLengths : Vect n String -> Vect n Nat
allLengths [] = []
allLengths (word :: words) = 0 :: allLengths words
```

所需的类型是 `Vect (S k) Nat`, 因此, 和以前一样, Idris 意识到唯一可能的结果是非空向量。它还意识到它可以通过在单词上递归调用 `allLengths` 来找到 `Vect k Nat` 类型的值。

对于向量头部的 `Nat`, Idris 找到了第一个满足类型的值 `0`, 但这并不是您想要的, 因此您可以将其替换为一个孔
`?vecthead:`

```
allLengths : Vect n String -> Vect n Nat
allLengths [] = []
allLengths (word :: words) = ?vecthead :: allLengths words
```

检查 `?vecthead` 的类型确认您正在寻找一个 `Nat`:

```
单词:字符串 k:Nat
words : Vect k 字符串
-----
vecthead : 纳特
```

和以前一样, 你可以通过用长度词填充这个洞来完成定义。所以, 更精确的类型不仅让你对程序的正确性更有信心, 在编写程序时给你更多的信息, 它也给了 Idris 一些信息, 让它为你写出好一点的程序。

3.2.3 类型、定义、细化: 对向量进行排序

对于本章到目前为止你编写的所有函数, 你都遵循了这个过程:

- 1 写一个类型。
- 2 创建骨架定义。
- 3 参数的模式匹配。
- 4 使用类型驱动表达式搜索和手动细化孔的组合填充右侧的孔。

但是, 通常还有一些工作要做。例如, 您可能会发现需要创建额外的辅助函数、检查中间结果或优化您最初为函数指定的类型。

您可以通过创建一个返回输入向量的排序版本的函数在实践中看到这一点。您可以使用插入排序, 这是一种简单的排序算法, 很容易以函数式风格实现, 非正式地描述如下: 给定一个空向量, 返回一个空向量。给定一个向量的头和尾, 对向量的尾进行排序, 然后插入

头进入排序的尾部, 以使结果保持排序。

为类型增加精度 : 使用向量

您可以交互地编写它 , 从清单中显示的骨架定义开始

3.3 打开一个 Atom 缓冲区并将此代码放入名为 VecSort.idr 的文件中。记住
您可以使用 Ctrl-Alt-A 从类型创建 insSort 的骨架定义。

清单 3.3 insSort 在具有初始类型的向量上的骨架定义(VecSort.idr)

导入数据.Vect

```
insSort : Vect n elem -> Vect n elem
insSort xs = ?insSort_rhs
```

此类型明确指出输出必须与输入具有相
同的长度。元素类型 elem 由类型级变量给
出 , 因此它代表任何类型。

在您完成此过程时 , 我建议您检查每个类型的
使用 Ctrl-Alt-T 出现的漏洞 , 并确保您了解
变量和漏洞。

开发工作流程打开 Atom 窗口通常很有用
用于交互式编辑文件 , 以及打开REPL的终端窗口
用于测试、评估、检查文档等。

编写了此函数的类型后 , 通过执行以下操作来实现它 :

1 定义使用 Ctrl-Alt-C 在 xs 上进行大小写拆分 , 导致 :

```
insSort : Vect n elem -> Vect n elem
insSort [] = ?insSort_rhs_1
insSort (x :: xs) = ?insSort_rhs_2
```

2 优化 - 尝试对 ?insSort_rhs_1 进行表达式搜索 , 结果如下 :

```
insSort : Vect n elem -> Vect n elem
插入排序 [] = []
insSort (x :: xs) = ?insSort_rhs_2
```

正如预期的那样 , 已排序的空向量本身就是一个空向量。

3 Refine - 不幸的是 , 尝试对 ?insSort_rhs_2 进行表达式搜索的结果更少
有效的 :

```
insSort : Vect n elem -> Vect n elem
插入排序 [] = []
insSort (x :: xs) = x :: xs
```

虽然 Idris 知道向量应该有多长 , 并且它有局部变量
在正确的类型中 , insSort 的整体类型对 Idris 来说不够精确
用你想要的程序填补这个空白。

表达式搜索 如本例所示 , 虽然表达式

搜索通常可以引导您找到有效功能 , 它不能代替了解程序的工作原理 ! 你需要了解算
法 , 但是
您可以使用表达式搜索来帮助填写详细信息。

4 定义 在递归数据类型上编写函数时,通常

有效地对结构的递归部分进行递归调用。这里,
您可以通过递归调用`insSort xs`对尾部进行排序并将结果绑定到
本地定义的变量:

```
insSort : Vect n elem -> Vect n elem
插入排序 [] = []
insSort (x :: xs) = let xsSorted = insSort xs in
    ?insSort_rhs_2
```

5 类型在?insSort_rhs_2 中,您要将头部x插入已排序的尾部,

`xs` 已排序。因为这会稍微复杂一些,所以您可以将
通过按住 Ctrl-Alt-L 并将光标悬停在顶层定义上
`?insSort_rhs_2`,导致以下结果:

```
insSort_rhs_2 : (x : elem) -> (xs : Vect k elem) ->
    (xsSorted : Vect k elem) ->
    Vect (S k) elem

insSort : Vect n elem -> Vect n elem
插入排序 [] = []
insSort (x :: xs) = let xsSorted = insSort xs in
    insSort_rhs_2 x xs xsSorted
```

这创建了一个具有新类型但没有实现的新顶级函数,并且它用对新函数的调用替换了漏洞。新函数的参数是在漏洞范围内的局部变量

`?insSort_rhs_2`。

6 精炼 一旦你意识到新函数的工作是将x插入
`xsSorted`向量,可以编辑`insSort_rhs_2`的名称和类型来体现
这个。您可以删除不需要的参数:

```
插入 : (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem

insSort : Vect n elem -> Vect n elem
插入排序 [] = []
insSort (x :: xs) = let xsSorted = insSort xs in
    插入 x xsSorted
```

提升定义 当使用 Ctrl-Alt-L 提升定义时,Idris 将使用生成类型中的所有局部变量生成与孔同名的新定义。在这种情况下,你知道你不需要

所有这些,因此您可以编辑掉不必要的`xs`参数。

7 定义 您现在必须定义插入。再次,创建一个骨架定义:

```
插入 : (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x xsSorted = ?insert_rhs
```

然后在`xsSorted` 上进行大小写拆分,导致:

```
插入 : (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = ?insert_rhs_1
插入 x (y :: xs) = ?insert_rhs_2
```

为类型增加精度：使用向量

8 优化 - 对 `insert_rhs_1` 的表达式搜索导致以下结果：

```
插入 : (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = [x]
插入 x (y :: xs) = ?insert_rhs_2
```

这是有效的，因为 Idris 知道它正在寻找一个包含一个元素的向量
类型 `elem`，而类型 `elem` 唯一可用的东西是 `x`。

9 Refine - 对于 `(y :: xs)` 情况，如果有孔 `?insert_rhs_2`，您就会遇到问题。有两种情况需要考虑：

如果 `x < y`，结果应该是 `x :: y :: xs`，因为结果不会是

如果 `x` 在 `y` 之后插入，则排序。

否则，结果应该以 `y` 开头，然后将 `x` 插入
尾巴 `xs`。

您的问题是您对元素类型 `elem` 一无所知。你需要
对其进行约束，以便您知道可以比较 `elem` 类型的元素。你
可以细化插入的类型（以及相应的 `insSort`），以便您可以
做必要的比较：

```
插入 : 单词元素 =>
      (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = [x]
插入 x (y :: xs) = ?insert_rhs_2

insSort : Ord elem => Vect n elem -> Vect n elem
插入 排序 [] = []
insSort (x :: xs) = 让 xsSorted = insSort xs in
                  插入 x xsSorted
```

请记住，在第 2 章中，您通过在类型中的 `=>` 之前放置诸如 `Ord elem` 之类的约束来约束泛型类型。你会看到更多关于这个的
第七章。

10 定义 您现在需要检查 `x < y` 并对结果采取行动。一种方法来做到这一点
带有 `if...then...else` 构造：

```
插入 : 单词元素 =>
      (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = [x]
插入 x (y :: xs) = 如果 x < y 则 x :: y :: xs
                           否则 y :: 插入 x xs
```

或者，您可以使用交互式编辑器为定义提供更多结构，并插入 `case` 构造以匹配中间结果。按

`Ctrl-Alt-M` 将光标悬停在 `?insert_rhs_2` 上。这引入了一个新案例
带有要检查的值的占位符的表达式（因此函数
不会进行类型检查）：

```
插入 : 单词元素 =>
      (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = [x]
```

```
插入 x (y :: xs) = 大小写
      _ 的
      case_val => ?insert_rhs_2
```

这 _ 代表您需要提供的表达式,以便函数
类型检查成功。您需要填写以匹配: _ 用你想要的表达

```
插入 :单词元素 =>
  (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = [x]
插入 x (y :: xs) = case x < y of
  case_val => ?insert_rhs_2
```

11 定义 您现在可以使用 Ctrl-Alt-C 以通常的方式对 case_val 进行大小写拆分,
导致:

```
插入 :单词元素 =>
  (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = [x]
插入 x (y :: xs) = case x < y of
  错误 => ?insert_rhs_1
  真 => ?insert_rhs_3
```

12 细化 最后,通过填充新的漏洞来完成实现

insert_rhs_1 和 insert_rhs_3。不幸的是,表达式搜索无济于事
你在这里很多,因为类型中没有足够的信息,所以你
需要手动填写:

```
插入 :单词元素 =>
  (x : elem) -> (xsSorted : Vect k elem) -> Vect (S k) elem
插入 x [] = [x]
插入 x (y :: xs) = case x < y of
  假 => y :: 插入 x xs
  真 => x :: y :: xs
```

定义完成后,您可以在REPL 中对其进行测试,如下所示:

```
*VecSort> insSort [1,3,2,9,7,6,4,5,8]
[1, 2, 3, 4, 5, 6, 7, 8, 9] : Vect 9 整数
```

记住整体检查!

不要忘记检查 insSort 是否为总计:

```
*VecSort> :总插入
Main.insSort 是总计
```

检查您定义的函数是否完整是一个好习惯。如果一个函数的类型正确,但不是全部,那么在您测试它时它可能会工作,但可能仍然存在
在一些不寻常的输入上是一个细微的错误,例如缺少模式或可能的非终止。

总而言之,按照 type-define-refine 过程,您已经完成了以下操作:

- 1为insSort编写一个类型。
- 2尝试定义insSort直到遇到插入的需要,然后将其提升到具有自己类型的新顶级定义。
- 3尝试定义插入,直到您遇到比较需要,此时您改进了类型以支持elem 上的排序约束。
- 4继续使用细化类型定义插入,从而完成
插入排序的实现。

练习



结束本节,这里有一些练习来测试你对List和Vect 的交互式编辑模式和模式匹配的理解。

以下函数或它们的一些变体在 Prelude 或

数据.Vect:

- 1长度:列出 a -> Nat 2反向:列出 a ->
列出 a
- 3地图:(a -> b) -> 列表 a -> 列表 b 4地图:(a -> b) -> Vect
na -> Vect nb

对于它们中的每一个,使用 Atom 中的交互式编辑定义您自己的版本。请注意,您需要使用不同的名称(例如 my_length、my_reverse、my_map)以避免与 Prelude 中的名称发生冲突。您可以在REPL上测试您的答案,如下所示:

```
*ex_3_2> my_length [1..10]
10: Nat

*ex_3_2> my_reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1] : 列表整数

*ex_3_2> my_map (* 2) [1..10]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20] : 列表整数

*ex_3_2> my_vect_map 长度 [ “热”、“狗”、“跳跃”、“青蛙” ]
[3, 3, 7, 4] : Vect 4 Nat
```

不要忘记检查您的定义是否完整!

3.3 示例:矩阵函数的类型驱动开发

与列表相反,您可能使用类型中显式长度的向量的主要原因是让向量的长度有助于更快地引导您找到工作函数。当您使用二维向量时,这可能特别有用。

这些反过来又有助于在矩阵上实现运算,这些矩阵在编程中有多种应用,例如 3D 图形。

在数学中,矩阵是按行和列排列的数字的矩形阵列。图 3.3 以两种数学符号显示了一个示例 3×4 矩阵,

并以 Idris 表示法作为向量的向量。请注意,当将矩阵表示为嵌套向量,矩阵的维度在类型中变得明确。

$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$	$\begin{bmatrix} [1, 2, 3, 4], \\ [5, 6, 7, 8], \\ [9, 10, 11, 12] \end{bmatrix}$	图 3.3 将矩阵表示为二维向量。在左边,矩阵是数学符号。相同 矩阵在右侧的 Idris 中表示。
3×4 矩阵	Vect 3 (Vect 4 Int)	矩阵在右侧的 Idris 中表示。

3.3.1 矩阵运算及其类型

在对矩阵进行运算 (例如加法和乘法)时,它是检查您正在使用的向量的尺寸是否适合操作非常重要。例如:

添加矩阵时,每个矩阵必须具有完全相同的维度。

加法是通过在每个矩阵中添加相应的元素来实现的。例如,您可以添加两个 3×2 矩阵,如下所示:

$$\begin{array}{ccc} 1 & 2 & 7 & 8 \\ 3 & 4 & + & 9 & 10 \\ & & = & & 8 & 10 \\ & & & & 12 & 14 \\ & & & & 11 & 12 \end{array}$$

以下将 2×2 矩阵与 3×2 矩阵相加是无效的,因为第三行没有对应的元素:

$$\begin{array}{ccc} 1 & 2 & 7 & 8 \\ 3 & 4 & + & 9 & 10 \\ & & = & ? & ? \end{array}$$

因此,矩阵加法的类型可能如下:

```
addMatrix : Num numType =>
    Vect 3 (Vect 2 numType) ->
    Vect 3 (Vect 2 numType) ->
    Vect 3 (Vect 2 numType)
```

换句话说,对于一些数值类型numType,添加一个rows \times cols矩阵到rows \times cols矩阵产生rows \times cols矩阵。

矩阵相乘时,左矩阵的列数必须为与右矩阵中的行数相同。然后,乘法工作如下:

示例：矩阵函数的类型驱动开发

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \end{pmatrix} = \begin{pmatrix} 29 & 32 & 35 & 38 \\ 65 & 72 & 79 & 86 \\ 101 & 112 & 123 & 134 \end{pmatrix}$$

结果的第 2 行第 3 列 = $3 \times 9 + 4 \times 13 = 79$

在这里，将 3×2 矩阵乘以 2×4 矩阵会得到 3×4 矩阵。这结果中 x 行、y 列中的值是左侧输入的 x 行和右侧输入的 y 列中相应元素的乘积之和。所以

矩阵乘法的类型可能如下：

```
multMatrix : Num numType =>
    Vect n (Vect m numType) -> Vect m (Vect p numType) ->
    Vect n (Vect p numType)
```

换句话说，对于某些数值类型 numType，将一个 $n \times m$ 矩阵乘以一个 $m \times p$ 矩阵产生一个 $n \times p$ 矩阵。

3.3.2 转置矩阵

处理矩阵时一个有用的操作是转置，它转换行列，反之亦然。例如，一个 3×2 矩阵变成一个 2×3 矩阵：

1 2	
<small>135</small>	
3 4	转换为 ...
	246
	5 6

您可以编写一个 transposeMat 函数，该函数通常将 $m \times n$ 矩阵转换为一个 $n \times m$ 矩阵，将矩阵表示为嵌套向量。像往常一样，您可以编写以交互方式运行，其中每个步骤被广泛地描述为类型、定义、或细化。从这一点开始，我通常假设您对 Atom 中的交互式命令感到满意，我将描述整个类型驱动的过程，而不是

构建功能的细节。

1 类型首先给出 transposeMat 的类型：

```
transposeMat : Vect m (Vect n elem) -> Vect n (Vect m elem)
```

对于矩阵运算，在 addMatrix 和 multMatrix 类型中，您需要将元素类型限制为数字。但是，这里的元素类型矩阵 elem 可以是任何东西。您不会检查它或使用它 transposeMat 实施中的任何一点；您只需更改行到列和列到行。

2 定义创建骨架定义,然后对输入向量进行大小写:

```
transposeMat : Vect m (Vect n elem) -> Vect n (Vect m elem)
transposeMat [] = ?transposeMat_rhs_1
transposeMat (x :: xs) = ?transposeMat_rhs_2
```

3 类型当 Idris 创建新的漏洞时,无论是从案例拆分还是表达式搜索的不完整结果,检查这些漏洞的类型以深入了解如何进行总是一个好主意。首先看一下?transposeMat_rhs_1的类型:

```
elem : 类型 n : Nat
```

```
-----
```

```
transposeMat_rhs_1 : Vect n (Vect 0 elem)
```

在这里,您尝试将一个 $0 \times n$ 向量转换为一个 $n \times 0$ 向量,因此您需要创建一个空向量的n个副本。我们稍后会回到这个案例;现在,您可以将孔重命名为createEmpties并使用 Ctrl-Alt-L 将其提升为顶级函数:

```
createEmpties : Vect n (Vect 0 elem)

transposeMat : Vect m (Vect n elem) -> Vect n (Vect m elem)
transposeMat [] = createEmpties transposeMat
transposeMat (x :: xs) = ?transposeMat_rhs_2
```

4 类型查看?transposeMat_rhs_2 的类型:

```
elem : 类型 n : Nat
```

```
x : 向量元素
k : 晚上
xs : Vect k (Vect n elem)
```

```
-----
```

```
transposeMat_rhs_2 : Vect n (Vect (S k) elem)
```

你有xs,它是一个 $k \times n$ 矩阵,你需要制作一个 $n \times (S k)$ 矩阵。

5 定义 这里你需要的一个见解是,从一个 $n \times k$ 矩阵构建一个 $n \times (S k)$ 矩阵可能更容易,因为至少有一个维度是正确的。

您可以通过转置xs来创建这样的矩阵:

```
transposeMat (x :: xs) = 让 xsTrans = transposeMat xs in
                           ?transposeMat_rhs_2
```

6 类型您还可以将?transposeMat_rhs_2提升为顶级函数,将其重命名为transposeHelper。这导致以下结果:

```
transposeHelper : (x : Vect n elem) -> (xs : Vect k (Vect n elem)) ->
                           (xsTrans : Vect n (Vect k elem)) -> Vect n (Vect (S k) elem)
```

transposeHelper的类型是根据您可以访问的局部变量的类型生成的: x、xs和xsTrans。它将这些变量作为输入,并产生一个Vect n (Vect (S k) elem)作为输出。

示例:矩阵函数的类型驱动开发

7类型 在这个阶段,最好更仔细地查看变量x、xs和xsTrans的类型,并尝试使用这些类型来可视化完成transposeMat所需的操作。

为了直观起见,我们取n = 4和k = 2。图 3.4 显示了一个原始的二维向量,形式为(x :: xs),其中x是第一行, xs是其余行,用这些尺寸建造。它识别组件x和xs,并显示转置xs的结果和整个操作的预期结果。

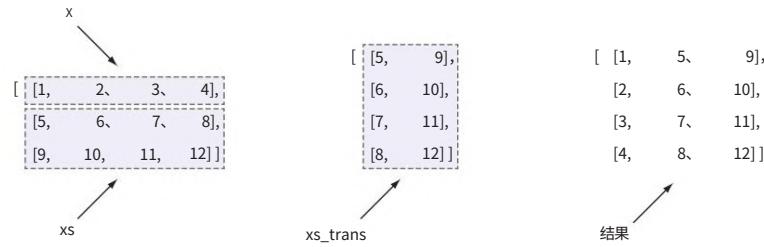


图 3.4 您正在转置的向量的分量 (x和xs) ,以及转置xs的结果,以及预期的总体结果

因此,您需要做的是将x的每个元素添加到xsTrans的相应元素中的向量中;你在transposeHelper中不需要xs。如果你在 transposeHelper 的类型和 transposeMat 中的应用程序中手动删除它,你会得到:

```
transposeHelper : (x : Vect n elem) -> (xsTrans : Vect n (Vect k elem)) ->
  Vect n (Vect (S k) 元素)

transposeMat : Vect m (Vect n elem) -> Vect n (Vect m elem)
transposeMat [] = createEmpties
transposeMat (x :: xs) = let xsTrans = transposeMat xs in transposeHelper x xsTrans
```

8定义 要实现transposeHelper,可以添加骨架定义,对x和xsTrans进行模式匹配,然后使用表达式搜索完成定义。有足够的信息让 Idris 自己填写详细信息:

```
transposeHelper : (x : Vect n elem) -> (xsTrans : Vect n (Vect k elem)) ->
  Vect n (Vect (S k) 元素)

transposeHelper [] [] = []
transposeHelper (x :: xs) (y :: ys) = transposeHelper xs ys
```

与其直接输入,不如尝试使用交互式命令构建它。可以仅使用 Ctrl-Alt-A、Ctrl-Alt-C、Ctrl-Alt-S 和光标移动从类型编写此函数。

向量上的大小写拆分 如果您先在x上拆分大小写,然后在 xsTrans 上拆分大小写,请注意 Idris 只为xsTrans 提供了一种可能的模式。这是因为x和xsTrans的类型表明两者必须具有相同的长度。

9 定义 剩下的就是实现`createEmpties`。您可以实施这使用库函数,复制:

```
*转置> :doc Vect.replicate
Data.Vect.replicate : (n : Nat) -> (x : a) -> Vect na
    重复某个值 n 次
    论据:
        n : Nat -- 重复的次数
        x : a -- 要重复的值
```

如果您从其类型创建`createEmpties`的骨架定义,您将看到下列的:

```
createEmpties : Vect n (Vect 0 elem)
createEmpties = ?createEmpties_rhs
```

您需要调用复制来构建一个包含n个空列表的向量。很遗憾,因为左侧的模式中没有可用的局部变量,自然定义导致错误消息:

```
createEmpties : Vect n (Vect 0 elem)
createEmpties = replicate n [] -- 没有这样的变量 n
```

问题是n是一个类型级别的变量,并且不能被`createEmpties`的定义访问。很快,在3.4节中,您将看到如何处理类型级别的变量,以及如何直接编写`createEmpties`。为了

此刻,因为类型表明只有一个有效值
要复制的长度参数,您可以使用下划线代替:

```
createEmpties : Vect n (Vect 0 elem)
createEmpties = 复制 _ []
```

`transposeMat`的实现现在已经完成。供参考,完整
定义见清单3.4。你可以在REPL上测试它:

```
*transpose> transposeMat [[1,2], [3,4], [5,6]]
[[1, 3, 5], [2, 4, 6]] : Vect 2 (Vect 3 整数)
```

清单 3.4 矩阵转置的完整定义 (Matrix.idr)

```
createEmpties : Vect n (Vect 0 elem)
createEmpties = 复制 _ []
transposeHelper : (x : Vect n elem) ->
    (xsTrans : Vect n (Vect k elem)) ->
    Vect n (Vect (S k) 元素)
    转置助手[] [] = []
    transposeHelper (x :: xs) (y :: ys) = (x :: y) :: transposeHelper xs ys
transposeMat : Vect m (Vect n elem) -> Vect n (Vect m elem)
transposeMat [] = createEmpties
transposeMat (x :: xs) = 让 xsTrans = transposeMat xs in
    transposeHelper x xsTrans
```

示例:矩阵函数的类型驱动开发

代码重用当使用

`type Define-Refine` 过程以交互方式构建定义时,最好注意可以使定义更通用的部分,或者可以使用现有库函数实现的部分。

例如, `transposeHelper` 的结构与库函数 `zipWith` 非常相似,它对两个向量中的对应元素应用一个函数,定义如下:

```
zipWith : (a -> b -> c) -> Vect n a -> Vect n b -> Vect n c
zipWith f [] = []
zipWith f (x :: xs) (y :: ys) = f x y :: zipWith f xs ys
```

练习



1 使用 `zipWith` 而不是 `transposeHelper` 重新实现 `transposeMat`。

您可以在REPL上测试您的答案,如下所示:

```
*ex_3_3_3> transposeMat [[1,2], [3,4], [5,6]]
[[1, 3, 5], [2, 4, 6]] : Vect 2 (Vect 3 整数)
```

2 实施 `addMatrix`: Num a => Vect n (Vect ma) -> Vect n (Vect ma) -> Vect n
(矢量马)。

您可以在REPL上测试您的答案,如下所示:

```
*ex_3_3_3> addMatrix [[1,2], [3,4]] [[5,6], [7,8]]
[[6, 8], [10, 12]] : Vect 2 (Vect 2 整数)
```

3 实现一个矩阵相乘函数,按照中给出的描述

第 3.3.1 节。

提示:这个定义非常棘手,涉及多个步骤。考虑以下:

你有一个维度为 $n \times m$ 的左矩阵和一个维度为 $m \times p$ 的右矩阵。

一个好的开始是在右矩阵上使用 `transposeMat`。

请记住,您可以使用 `Ctrl-Alt-L` 将孔提升到顶级函数。记住要密切注意局部变量的类型和类型

的孔。

记得使用 `Ctrl-Alt-S` 搜索表达式,并密切关注

任何产生的孔的类型。

您可以在REPL上测试您的答案,如下所示:

```
*ex_3_3_3> multMatrix [[1,2], [3,4], [5,6]] [[7,8,9,10], [11,12,13,14]] [[29,32,35,38], [65,72,79,86],
```

```
[101,112,123,134]] : Vect 3 (Vect 4 整数)
```

3.4 隐式参数:类型级别的变量

您现在已经看到了几个类型级别的变量定义,这些变量可以代表类型或值。例如:

反向:列出元素 -> 列出元素

这里, elem是一个类型级变量,代表列表的元素类型。它出现了两次,在输入类型和返回类型中,所以每个元素类型必须是相同的。

附加:Vect n elem -> Vect m elem -> Vect (n + m) elem

这里, elem又是一个类型级变量,代表向量的元素类型。n和m是代表输入向量长度的类型级变量,它们在输出中再次用于描述输出长度与输入长度的关系。

这些类型级别的变量没有在其他任何地方声明。因为类型是一等的,所以类型级别的变量也可以被引入范围并在定义中使用。这些类型级别的变量被称为函数reverse和append的隐式参数。在本节中,您将看到隐式参数如何工作,以及如何在定义中使用它们。

3.4.1 隐含参数的需要

为了说明隐式参数的必要性,让我们看一下如何在没有它们的情况下定义append。您可以将elem、n和m参数设置为显式附加,从而得到以下定义:

```
附加: (elem:类型) -> (n:Nat) -> (m:Nat) ->
      Vect n elem -> Vect m elem -> Vect (n + m) elem
append elem Z m [] ys = ys append elem (S k)
m (x :: xs) ys = x :: append elem km xs ys
```

但是如果你这样做了,在调用append时你还必须明确元素类型和长度:

```
*Append_expl> append Char 22[ a , b ][ c , d ][ a , b , c , d ]:
Vect 4 Char
```

给定参数[a , b]和[c , d]的类型,每个参数elem (必须是Char)、n (必须是2,从[a , b]的长度开始),和m (也必须是2,从[c , d]的长度开始)。其中任何一个的任何其他值都不会被很好地键入。

因为向量参数的类型中有足够的信息,所以Idris可以推断a、n和m参数的值。因此,您可以这样写:

```
*Append> append__ [ a , b ][ c , d ][ a , b , c ,
d ]: Vect 4 Char
```

隐式参数:类型级变量

隐含值

函数调用中的下划线(_)表示您希望 Idris 计算出参数的隐式值,给定表达式其余部分中的信息:

```
*Append> append____[ a , b ][ c , d ][ a ,
b , c , d ]:Vect 4 Char
```

如果不能,Idris 会报错:

```
*Append> append____[ c , d ](input): 不能推断参数
n 追加,
无法推断要附加的显式参数
```

在这里,Idris 报告说它无法计算出第一个向量的长度,或者第一个向量本身的长度。与代表尚未编写的部分表达式的孔不同,在分数下代表只有一个有效值的表达式部分。如果 Idris 无法推断下划线的唯一值,则这是一个错误。

使用隐式参数避免了需要显式编写可以由 Idris 推断的详细信息。将 elem 、 n 和m隐含在append的类型中意味着您可以在需要时直接在类型中引用它们,而无需在调用函数时给出显式值。

3.4.2 绑定和非绑定隐式

让我们再看一下带有隐式参数的reverse和append的类型:

反向:列出元素 -> 列出元素

附加:Vect n elem -> Vect m elem -> Vect (n + m) elem

名称elem 、 n 和m称为未绑定隐式。这是因为它们的名称是直接使用的,没有在其他任何地方声明 (或绑定) 。您也可以将这些类型编写如下:

```
reverse : {elem : Type} -> List elem -> List elem
append : {elem : Type} -> {n : Nat} -> {m :
Nat} ->
Vect n elem -> Vect m elem -> Vect (n + m) elem
```

在这里,隐式参数已显式绑定在类型中。符号{x : S} -> T表示函数类型,其中参数旨在由 Idris推断,而不是由程序员直接编写。

当您编写具有未绑定隐式的类型时,Idris 将查找未定义的名称
在类型中,并在内部将它们转换为绑定的隐式。考虑这个例子:

附加:Vect n elem -> Vect m elem -> Vect (n + m) elem

首先,Idris 识别出elem 、 n 和m是未定义的,因此它在内部将类型重写如下:

```
附加:{elem:_}>{n:_}>{m:_}>
Vect n elem -> Vect m elem -> Vect (n + m) elem
```

请注意,它没有尝试为新参数填充类型,而是将它们作为下划线给出,希望它能够从类型的其余部分中的一些其他信息中推断出它们。在这里,这将导致以下结果:

```
附加:{elem:类型} -> {n:Nat} -> {m:Nat} ->
Vect n elem -> Vect m elem -> Vect (n + m) elem
```

未绑定的隐式名称在实践中,

Idris 不会将每个未定义的名称视为未绑定的隐式名称 仅以小写字母开头并且单独出现或出现在函数参数位置的名称。鉴于以下情况,

测试: fma -> b -> a

m 出现在参数位置, a 也是如此。b单独出现, f只出现在函数位置。因此, m、a 和b被视为未绑定的隐含其。f不被视为未绑定的隐式,这意味着必须在其他地方定义它才能使此类型有效。

通常,为了简洁起见,您将隐式保留为未绑定,但在某些情况下,使用绑定隐式来代替很有用:

为了清晰和可读性,将显式类型赋予隐式是很有用的

论据。

如果隐式参数和其他参数之间存在依赖关系,您可能需要使用绑定隐式来让 Idris 清楚这一点。

3.4.3 在函数中使用隐式参数

在内部,Idris 像对待任何其他参数一样对待隐式参数,但是程序员不需要显式提供它们的符号方便。因此,您可以在函数定义中引用隐式参数,甚至可以对其进行大小写拆分。

例如,如何找到向量的长度?您可以按情况执行此操作
分割向量本身:

```
长度 : Vect n elem -> Nat 长度 [] = Z 长度 (x :: xs)=1+
长度 xs
```

因为长度是类型的一部分,你也可以直接引用它:

```
长度 : Vect n elem -> Nat 长度 {n} xs = n
```

模式中的符号{n}将隐式参数n带入范围,允许您直接使用它。

更一般地说,您可以通过使用
表示法{n = value},其中n是隐式参数的名称:

```
*Append> append {elem = Char} {n = 2} {m = 3} append : Vect 2 Char -> Vect 3
Char -> Vect 5 Char
```

在这里,您仅部分地将append应用于其隐式参数,提供了一个专门的函数,用于将两个字符的向量附加到三个字符的向量。

这个符号也可以用在定义的左侧,以对隐式参数进行大小写分割。例如,要实现3.3.2节中的createEmpties ,您可以直接将其写入

范围:

```
createEmpties : Vect n (Vect 0 a) createEmpties {n = ?}
createEmpties_rhs
```

如果你在Atom中对n进行大小写拆分,你会看到:

```
createEmpties : Vect n (Vect 0 a) createEmpties {n = Z}
= ?createEmpties_rhs_1 createEmpties {n = (S k)} = ?createEmpties_rhs_2
```

最后,您可以使用表达式搜索剩余的两个孔来完成定义:

```
createEmpties : Vect n (Vect 0 a) createEmpties {n = Z} =
[] createEmpties {n = (S k)} = [] :: createEmpties
```

请注意,在递归调用中, createEmpties就足够了。无需为长度提供明确的值,因为只有一个值(k)会进行类型检查。另一方面,如果您在没有长度值的情况下尝试REPL ,Idris将报告

一个错误:

```
*transpose> createEmpties (输入) :无法将
参数 n 推断为 createEmpties,
无法将参数 a 推断为 createEmpties
```

您可以通过为n和elem提供显式值或为表达式提供目标类型来解决此问题:

```
*转置> createEmpties {a=Int} {n=4}
[],[],[],[] : Vect 4 (Vect 0 Int)
```

```
*转置> (Vect 4 (Vect 0 Int)) createEmpties
[],[],[],[] : Vect 4 (Vect 0 Int)
```

类型和参数擦除因为隐式参数

内部被视为与任何其他参数相同,您可能想知道在运行时会发生什么。通常,您使用隐式参数为程序提供精确的类型,那么这是否意味着类型信息必须在运行时编译并呈现?

幸运的是,Idris编译器意识到了这个问题。它将在编译程序之前对其进行分析,以便在运行时不会出现任何仅用于类型检查的参数。

3.5 总结

Idris 中的函数由模式匹配方程的集合定义。 模式源于数据类型的构造函数。 Atom 文本编辑器提供交互式编辑模式,使用类型来帮助指导功能实现。 Emacs 和 Vim 也有类似的方式。

交互式编辑命令提供了自然的工具来跟踪编辑过程
类型,定义,细化。

交互式编辑提供搜索有效表达式的命令
满足孔的类型。

更精确的类型,例如Vect,为编译器提供更多信息,以帮助检查函数是否正确,并帮助约束
表达式搜索。

矩阵是二维向量,其维度编码在类型中。
诸如加法和乘法之类的矩阵运算可以被赋予精确描述运算如何影响维度的类型。

type-define-refine 过程通过使用类型来指导每个子表达式的实现并创建适当的辅助函数,帮助
您实现具有精确类型的矩阵运算。

类型级变量是函数的隐式参数,可以通过将它们括在大括号{} 中来将其带入范围并像任何其他参数
一样使用。

用户定义的数据类型

本章涵盖

定义自己的数据类型理解不同形式
的数据类型以类型驱动风格编写更大的交互式程序

类型驱动开发不仅涉及为函数提供精确的类型,如您目前所见,还涉及准确考虑数据的结构。从某种意义上说,编程(尤其是纯函数式编程)是将数据从一种形式转换为另一种形式。类型使我们能够描述该数据的形式,并且我们对这些描述进行得越精确,该语言在实现对该数据的转换时提供的指导就越多。

几种有用的数据类型作为 Idris 库的一部分分发,其中许多我们到目前为止都使用过,例如 List、Bool 和 Vect。除了直接在 Idris 中定义之外,这些数据类型没有什么特别之处。在任何实际的程序中,您都需要定义自己的数据类型来捕获您正在解决的问题的特定需求以及您正在使用的特定数据形式。不仅如此,仔细考虑数据类型的设计还有很大的好处:类型越精确地捕捉到问题的需求,您从交互式类型导向编辑中获得的好处就越大。

因此,在本章中,我们将研究如何定义新的数据类型。您将在许多小示例函数中看到各种形式的数据类型。我们还将开始研究一个更大的示例,即交互式数据存储,我们将在接下来的章节中对其进行扩展。首先,我们将只存储字符串,通过整数索引访问它们,但即使在这个小示例中,您也会看到用户定义类型和依赖类型如何帮助构建交互式接口,安全地处理可能的运行时错误。

4.1 定义数据类型

型构造函数和一个或多个数据构造函数定义。事实上,当您使用:doc查看数据类型的详细信息时,您已经看到了这些内容。例如,以下清单显示了:doc List 的输出,注释以突出显示类型和数据构造函数。

清单 4.1 来自:doc 的类型和数据构造函数

```
数据类型 Prelude.List.List : (elem : Type) -> Type
    通用列表

    构造函数:
        无:列出元素
        空列表

        (::) : 元素 -> 列表元素 -> 列表元素
            一个非空列表,由一个头元素和列表的其余部分组成。

    数据构造函数 Nil 不接受任何参数并
    返回一个空列表。
    数据构造函数 (::) 接受两个参数并返回
    一个列表。
```

List 的类型构造函数有一个
函数类型,它接受一个 Type
作为输入并返回一个 Type。

使用数据构造函数是构建类型构造函数给出的类型的规范方法。在List的情况下,这意味着具有List elem形式的每个值要么是Nil,要么对于某些元素x采用 $x :: xs$ 的形式,其余的

列表的 der, xs 。

我们将类型分为五个基本组,尽管它们都使用相同的语法定义:

枚举类型 通过直接给出可能值定义的类型联合类型 每个值携带附加数据的枚
举类型递归类型 根据自身定义的联合类型泛型类型 在某些其他类型上参数化的类
型依赖类型 从其他值计算出来的类型

在本节中,您将看到如何定义枚举类型、联合类型、递归类型和泛型类型;我们将在下一节讨论依赖类型。如果您以前使用函数式语言或任何允许您定义泛型类型的语言进行编程,那么我们在本节中讨论的类型应该是熟悉的,即使符号是新的。

命名约定 按照约定,我将使用首字母大写
类型构造函数和数据构造函数,以及一个初始的小写字母
功能。没有要求这样做,但它为代码的读者提供了有用的视觉指示。

4.1.1 枚举

枚举类型是通过提供该类型的有效值来直接定义的。最简单的例子是Bool,在 Prelude 中定义如下:

数据布尔 = 假 | 真的

要定义一个枚举数据类型,你需要给出data关键字来引入声明,然后给出类型构造函数的名称(在本例中为Bool)并列出
数据构造函数的名称(在本例中为True和False)。

图 4.1 显示了另一个例子,定义了一个枚举类型来表示
指南针上的四个基点。



图 4.1 定义方向
数据类型 (Direction.idr)

数据构造函数名称用竖线|分隔,没有限制

关于声明的布局方式(除了所有声明都在完全相同的列中开始的通常布局规则)。例如,它们可能各自位于不同的行:

```
数据方向 = 北
    | 东方
    | 南
    | 西方
```

一旦定义了数据类型,就可以使用它以交互方式定义函数。为了
例如,您可以如下定义一个turnClockwise函数,使用通常的 type、define、refine 过程:

1 Type 编写一个以Direction为输入和输出的函数类型,然后
创建骨架定义:

```
顺时针方向:方向 -> 方向
turnClockwise x = ?turnClockwise_rhs
```

2 定义- 通过x上的大小写拆分定义函数:

```
顺时针方向:方向 -> 方向
turnClockwise North = ?turnClockwise_rhs_1
```

```
turnClockwise East = ?turnClockwise_rhs_2
turnClockwise South = ?turnClockwise_rhs_3
turnClockwise West = ?turnClockwise_rhs_4
```

3 Refine 填充右侧的孔：

```
顺时针方向: 方向 -> 方向
顺时针北 = 北
顺时针东 = 南
顺时针南 = 西
顺时针西 = 北
```

4.1.2 联合类型

联合类型是枚举类型的扩展，其中的构造函数

`type` 本身可以携带数据。例如，您可以创建一个枚举类型
对于形状：

数据形状 = 三角形 | 矩形 | 圆圈

您可能希望使用形状存储更多信息，以便您可以绘制它，例如
例如，或计算其面积。这些信息会因形状而异：对于三角形，您可能想知道其底边的长度和高度。

对于一个矩形，您可能想知道它的长度和宽度。

对于一个圆，您可能想知道它的半径。

为了表示这些信息，每个数据构造函数，`Triangle`、`Rectangle`、
和`Circle`，可以给定参数类型，使用`Doubles`携带此数据来表示维度。图 4.2 显示了一些示例形状及其表
示。

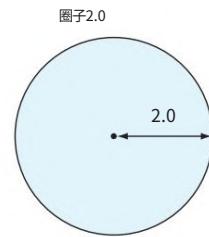
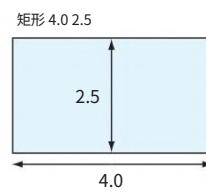
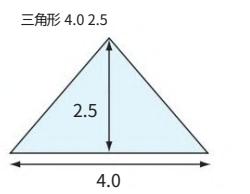


图 4.2 表示为联合类型的示例形状。三角形需要两个
`Doubles`，用于底数和高度；长方形

需要两个双精度数，宽度和高度；`Circle`采用`Double`
作为半径。

您可以将此形式的形状表示为 Idris 数据类型：

数据形状=三角形双双

```
| 长方形 双 双
| 圆形双人间
```

清单 4.2 展示了如何定义一个面积函数来计算每种可能性的形状面积。作为一个练习,不要直接输入这个函数,而是尝试使用 Atom 中的交互式编辑工具来构建它。

清单 4.2 定义Shape类型并计算其面积 (Shape.idr)

```
数据形状=三角形双双
|长方形 双 双
|圆形双人间

area : Shape -> Double area (Triangle
base height) = 0.5 * base * height area (Rectangle length height) = length * height
area (Circle radius) = pi * radius * radius
```

pi 在 Prelude 中被定义为一个常数。

如果您使用:doc查看Shape的文档,您可以看到此数据声明如何转换为类型和数据构造函数:

```
*Shape> :doc Shape 数据类型
Main.Shape : 类型
```

构造函数:

三角形:双 -> 双 -> 形状

矩形:双 -> 双 -> 形状

圆:双 -> 形状

当您定义新的数据类型时,使用文档注释提供将与:doc一起显示的文档也是一个好主意。在这种情况下,它有助于指示每个Double的用途。文档注释在数据声明中列出

通过在每个构造函数之前给出注释:

```
|||表示形状数据 Shape = |||一个三角
形,它的底长和高
    三角双双
    |||一个长方形,有它的长和高
        长方形 双 双
    |||一个圆,它的半径
        圆形双人间
```

使用 :doc 呈现如下:

```
*Shape> :doc Shape 数据类型
Main.Shape : 类型
    表示形状

构造函数:
    三角形:双 -> 双 -> 形状
        一个三角形,它的底长和高

    矩形:双 -> 双 -> 形状
        一个长方形,有它的长和高

    圆:双 -> 形状
        一个圆,它的半径
```

数据类型语法

明有两种形式。如您所见,其中之一列出了数据构造函数及其参数的类型:

数据形状=三角形双双

```
|长方形 双 双
圆形双人间
```

也可以通过直接给出数据类型和数据构造函数来定义数据类型,格式如 :doc 所示。您可以按如下方式定义 Shape:

数据形状:类型在哪里

```
三角形:双 -> 双 -> 形状
矩形:双 -> 双 -> 形状
圆:双 -> 形状
```

这与前面的声明相同。在这种情况下,它有点冗长,但这种语法更加通用和灵活。当我们定义依赖类型时,你很快就会看到更多。

我将在整本书中使用这两种语法。一般来说,我会使用简洁的形式,除非我需要额外的灵活性。

4.1.3 递归类型

类型也可以是递归的,即根据自身定义。例如, Nat在 Prelude 中递归定义如下:

日期 Nat = Z | 纳特

Prelude 还定义了函数和符号,以允许Nat像任何其他数字类型一样使用,因此您可以简单地写4,而不是写S (S (S (SZ)))。尽管如此,它的原始形式是使用定义的数据构造函数。

Nat 和效率

考虑到 Nat 是根据构造函数定义的,因此关注 Nat 的效率是合理的。不过,无需担心,原因有以下三个:

在实践中, Nat主要用于结构上,以类型来描述数据结构的大小,例如Vect ,因此Nat的大小对应于数据结构本身的大小。

编译程序时,类型会被擦除,因此出现在

类型级别,例如Vect 的长度,被擦除。

在内部,编译器优化Nat的表示,使其真正存储为机器整数。

您可以使用递归类型将上一节中的Shape示例扩展为
代表更大的图片。我们将图片定义为以下之一：原始形状

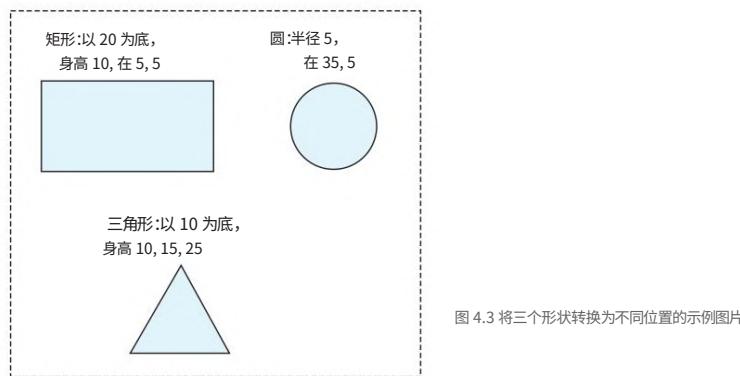
- 其他两张图片的组合
- 旋转一个角度的图片
- 一张图片被翻译到不同的位置

请注意，其中三个是根据图片本身定义的。你可以定义
跟在前面的非正式描述之后的图片类型，如下所示。

**清单 4.3 递归定义一个Picture类型,包括Shapes和更小的
图片(Picture.idr)**



图 4.3 显示了您可以使用此数据表示的图片类型的示例
类型。对于每个原始形状，我们将其位置视为
一个限定形状的假想盒子。



我们知道有三个子图，所以要在代码中表示这个，你可以开始
(定义)通过使用Combine将三个子图片放在一起：

```
测试图片:图片
testPicture = 合并 ?pic1 (合并 ?pic2 ?pic3)
```

继续，你知道每个子图片都被翻译到一个特定的位置，所以你
可以填充这些细节（细化），为原始形状本身留下孔：

```
测试图片 : 图片
testPicture = 组合 (翻译 5 5 ?rectangle)
  (合并 (翻译 35 5 ?圆)
    (翻译 15 25 ? 三角形) )
```

最后,您可以填充(细化)各个原始形状的细节。一种方法是在 Atom 中使用 Ctrl-Alt-L 将孔提升到顶层(类型),然后填写定义,从而得到以下最终定义。

清单 4.4 图 4.3 中的图片用代码表示 (Picture.idr)

```
矩形 : 图片 矩形 = 原始 (矩形 20 10)
```

```
圆 : 图片 圆 = 原始 (圆 5)
```

```
三角形 : 图片 三角形 = 原始 (三角形 10 10)
```

```
testPicture : 图片 testPicture = 组合 (平移 5 5 矩形)
  (合并 (翻译 35 5 圆)
    (翻译 15 25 三角形) )
```

因为 Rectangle 是 Shape 而不是 Picture 的数据构造函数,所以需要用 Primitive 构建图片。

像往常一样,您通过大小写拆分在图片数据类型上编写函数。要编写一个计算图片中每个原始形状面积的函数,您可以从编写一个类型开始:

图片区域 : 图片 -> 双

然后,创建一个骨架定义并对其参数进行大小写拆分。您应该达到以下目标:

```
pictureArea : 图片 -> 双图片区域 (Primitive x) = ?
pictureArea_rhs_1 pictureArea (组合 xy) = ?pictureArea_rhs_2 pictureArea
(旋转 xy) = ?pictureArea_rhs_3 pictureArea (平移 xyz) = ?pictureArea_rhs_4
```

这为您提供了一个大纲定义,向您展示了输入可以采用的形式并在右侧给出了孔。

Idris 在为图片区域创建模式时选择的变量 x,y 和 z 的名称并不是特别有用。您可以使用 %name 指令告诉 Idris 如何选择更好的默认名称:

```
%name 形状 shape, shape1, shape2 %name 图片 pic,
pic1, pic2
```

现在,当 Idris 需要为 Shape 类型的变量选择变量名时,它会默认选择 shape,如果 shape 已经在范围内,则选择 shape1,然后选择 shape2。同样,它会为 Picture 类型的变量选择 pic、pic1 或 pic2。

添加%name指令后,参数上的大小写拆分将导致模式具有更多信息的变量名称:

```
pictureArea : Picture -> Double pictureArea (Primitive
shape) = ?pictureArea_rhs_1 pictureArea (组合 pic pic1) = ?pictureArea_rhs_2 pictureArea
(Rotate x pic) = ?pictureArea_rhs_3 pictureArea (Translate xy pic) = ?pictureArea_rhs_4
```

完整的定义在清单 4.5 中给出。对于结构中遇到的每张图片,递归调用pictureArea,遇到Shape时,调用之前定义的area函数。

清单 4.5 计算图片中所有形状的总面积(Picture.idr)

<p>使用前面定义的面积函数来计算原始形状的面 积</p> <pre>pictureArea : Picture -> Double pictureArea (Primitive shape) = area shape pictureArea (组合 pic pic1) = pictureArea pic + pictureArea pic1 pictureArea (Rotate x pic) = pictureArea pic pictureArea (平移xy pic) = pictureArea pic</pre>	<p>当两张图片合并时,面积是这些图片的面积之和。</p>
<p>旋转图片时,该区域即为旋转后图片的面 积。</p>	<p>翻译图片时,该区域即为翻译后图片的区域。</p>

在REPL 测试结果定义总是一个好主意:

*图片>图片区域测试图片328.5398163397473:双

无限递归类型

像递归函数一样,需要至少一种非递归情况才能有用,因此至少有一个构造函数需要有一个非递归参数。如果您不这样做,您将永远无法构造该类型的元素。例如:

数据无限 = 永远无限

然而,使用泛型 Inf 类型来处理无限的数据流是可能的,我们将在第 11 章进行探讨。

4.1.4 通用数据类型

泛型数据类型是在某些其他类型上参数化的类型。就像您在第 2 章中看到的泛型函数类型一样,泛型数据类型允许您捕获常见的数据模式。

为了说明对泛型数据类型的需求,考虑一个返回上一节中定义的图片中最大三角形的面积。首先,您可以编写以下类型:

最大三角形:图片 \rightarrow 双

但是如果图片中没有三角形,它应该返回什么?你可以回来某种哨兵值,例如负尺寸,但这会违背类型驱动开发,因为您将使用Double来表示不是一个真正的数字。Idris 也没有空值。相反,您可以细化最大三角形的类型,引入了一个新的联合类型来捕获没有三角形的可能性:

数据最大 = 无三角形 | 双人间

最大三角形:图片 \rightarrow 最大

我将把最大三角形的定义留作练习。

您可能还想编写一个表示失败可能性的类型。例如,您可以为Double编写一个安全除法函数,如果该函数返回错误除以零:

```
数据 DivResult = DivByZero | 结果双倍
safeDivide : 双  $\rightarrow$  双  $\rightarrow$  DivResult
safeDivide xy = 如果 y == 0 则 DivByZero
                否则结果 (x / y)
```

Biggest和DivResult具有相同的结构!而不是定义多个这种形式的类型,您可以定义一个单一的泛型类型。其实这样的泛型存在于前奏曲中,叫做Maybe。以下清单显示了Maybe 的定义。

清单 4.6 捕获失败可能性的泛型类型 Maybe

```
数据也许 valtype =
    没有任何迹象表明 没有什么
    没有价值是 | 只是 valtype
    存储。          |
```

这里的名称 valtype 是一个类型级别的变量,代表您希望与 Maybe 一起使用的任何类型。

Just 是一个构造函数,它接受一个参数并指示存储单个值。

在泛型类型中,我们这里使用类型变量如valtype来代表具体类型。您现在可以使用Maybe Double而不是DivResult 定义safeDivide ,用Double实例化valtype :

```
safeDivide : 双  $\rightarrow$  双  $\rightarrow$  也许双
safeDivide xy = if y == 0 then Nothing
                否则只是 (x / y)
```

列表的定义

我们已经用泛型类型 List 编写几个函数,它定义为在前奏曲中如下:

```
数据列表 elem = Nil | (:) elem (列出 elem)
```

通用数据类型可以有多个参数,如图 4.4 所示要么,它在 Prelude 中定义,代表两种替代类型之间的选择。



通用类型和术语

您可能会听到人们非正式地提到“List 类型”或“Maybe 类型”。但是,这样做并不完全准确。列表本身不是一种类型,因为您可以在 REPL 确认:

```
伊德里斯> :t 列表
列表 :类型 -> 类型
```

从技术上讲,List 有一个函数类型,它以 Type 作为参数并返回一种。尽管 List Int 是一种类型,因为它已应用于具体参数,但 List 本身不是。相反,我们将非正式地将 List 称为泛型类型。

泛型类型的一个有用示例是二叉树结构。以下清单显示二叉树的定义,使用%name 指令给出命名提示以交互方式构建定义。

清单 4.7 定义二叉树 (Tree.idr)

```
数据树元素 = 空
          | 节点 (树元素)元素 (树元素)
%name 树树,tree1
```

←—— 没有数据的树
具有左子树、值和右子树的节点

二叉树通常用于存储有序信息，其中所有内容都在一个节点的左子树小于该节点的值，并且节点的所有内容右子树大于节点处的值。

这样的树被称为二叉搜索树，你可以写一个函数来插入一个将值放入这样的树中，前提是您可以对这些值进行排序：

```
插入:Ord elem => elem -> Tree elem -> Tree elem
插入 x 树 = ?insert_rhs
```

要编写这个函数，创建一个包含清单 4.7 中定义的 Tree.idr 文件，并执行以下操作：

1 定义 树上的案例拆分，给出树为空和其中树是一个节点：

```
插入:Ord elem => elem -> Tree elem -> Tree elem
插入 x 空 = ?insert_rhs_1
插入 x (节点树 y tree1) = ?insert_rhs_2
```

即使使用%name 指令，名称树、y 和tree1 也不是特别重要信息丰富，所以让我们重命名它们以表明它们是左子树，节点和右子树的值分别为：

```
插入:Ord elem => elem -> Tree elem -> Tree elem
插入 x 空 = ?insert_rhs_1
插入 x (节点左 val 右) = ?insert_rhs_2
```

2 Refine - 对于?insert_rhs_1，您创建一个具有空子树的新树节点：

```
插入:Ord elem => elem -> Tree elem -> Tree elem
插入 x 空 = 节点空 x 空
插入 x (节点左 val 右) = ?insert_rhs_2
```

3 定义 对于?insert_rhs_2，您需要比较要插入的值 x，与节点处的值，val。如果 x 小于 val，则将其插入左侧子树。如果相等，则它已经在树中，因此您将树原样返回。如果它大于 val，则将其插入右子树。为此，您可以使用 Prelude 中的 compare 函数，它返回一个元素排序枚举：

```
数据排序 = LT | EQ | GT
比较:Ord a => a -> a -> Ordering
```

您将对 compare x val 的中间结果执行匹配。在 insert_rhs_2 上按 Ctrl Alt-M：

```
插入:Ord elem => elem -> Tree elem -> Tree elem
插入 x 空 = 节点空 x 空
插入 x (Node left val right) = case
  _ 的
    case_val => ?insert_rhs_2
```

4 定义 case 表达式在要测试的表达式之前不会进行类型检查，因此将其 _ 替换为 compare x val：

定义数据类型

```
插入 :Ord elem => elem -> Tree elem -> Tree elem
insert x Empty = Node Empty x Empty insert x (Node left val
right) = case compare x val of
            case_val => ?insert_rhs_2
```

5 定义 如果现在对case_val 进行大小写拆分,您将获得LT、 EQ和

```
插入 :Ord elem => elem -> Tree elem -> Tree elem
insert x Empty = Node Empty x Empty insert x (Node left val
right) = case compare x val of
            LT => ?insert_rhs_1 EQ => ?
            insert_rhs_3 GT => ?insert_rhs_4
```

下面的清单显示了这个函数的完整定义,在改进了剩余的漏洞之后。

清单 4.8 将值插入二叉搜索树 (Tree.idr)

```
insert : Ord elem => elem -> Tree elem -> Tree elem insert x Empty = Node Empty x Empty
insert x (Node left val right)
= 案例比较 x val
    LT => 节点 (插入 x 左) val 右
    EQ => 节点左 val 右
    GT => 节点左值 (插入 x 右)

x 已经在树中,因为它等于 val,所以
返回原始树。                                     x 小于节点处的 val,因此返回
                                                    一棵新树,其中 x 插入到左子
                                                    树中。
                                                    <-->
                                                    <-->
```

@patterns 在

insert 中,你可能已经注意到,在EQ分支中,你返回的值和左边的pattern完全一样。为方便起见,您还可以命名模式:

```
插入 x orig@(Node left val right) = case compare x val of
    LT => 节点 (插入 x 左) val 右
    EQ => 原点
    GT => 节点左值 (插入 x 右)
```

符号 orig@ (Node left val right) 将名称 orig 赋予模式 Node left val right。它不会改变模式匹配的含义,但它确实意味着您可以在右侧使用名称 orig 而不是重复模式。

插入类型中的泛型变量elem需要有Ord约束,否则您将无法使用比较。替代

方法是在树类型本身中捕获Ord约束,改进

键入以包含此额外精度。下面的清单显示了如何通过直接给出类型和数据构造函数来做到这一点。

清单 4.9 在类型 (BSTree.idr) 中具有排序约束的二叉搜索树

将此类型命名为 BSTree 而不是
树,因为该类型是明确的
表示二叉搜索树。

对数据设置 Ord 约束
构造函数,所以你不能有一个搜索树
包含未排序的值。

→ 数据 BSTree : 类型 -> 类型在哪里
空:Ord elem => BSTree elem
节点:Ord elem => (左:BSTree elem) -> (val:elem) ->
(右:BSTree elem) -> BSTree elem

插入:元素 -> BSTree 元素 -> BSTree 元素
插入 x 空 = 节点空 x 空
插入 x orig@(节点左 val 右)
= 案例比较 x val
LT => 节点 (插入 x 左) val 右
EQ => 原点
GT => 节点左值 (插入 x 右)

不需要对插入进行 Ord 约束,因为
在 elem 中已经存在 Ord 约束

BSTree 本身。

精确和重用在树结构本身中放置一个约束会使
类型更精确,因为它现在只能存储可以在节点上比较的值,但代价是降低了可重用性。这是定义
新数据类型时经常需要考虑的权衡。有多种方法可以管理这种权衡,例如将数据与谓词配对

描述数据的形式,你将在第 9 章看到。

练习

1 编写一个函数listToTree : Ord a => List a -> Tree a, 将列表的每个元素插入二叉搜索树。

您可以在REPL上进行如下测试：

```
*ex_4_1> listToTree [1,4,3,5,2]
节点 (节点空 1 空)
2
(Node (Node Empty 3 (Node Empty 4 Empty))
5
空) :树整数
```

2 编写相应的函数treeToList : Tree a -> List a, 将一棵树展平
使用中序遍历进入一个列表 (即一个节点的左子树中的所有值
应该在节点的值之前添加到列表中,应该添加
在右子树中的值之前)。

如果你对练习 1 和 2 有正确的答案,你应该能够跑
这个:

```
*ex_4_1> treeToList (listToTree [4,1,8,7,2,3,9,5,6])
```

[1, 2, 3, 4, 5, 6, 7, 8, 9] : 列表整数

3 整数算术表达式可以采用以下形式之一：

单个整数表达式与表
达式的加法 表达式与表达式的减法 表达式与表达式
的乘法

定义可用于表示此类表达式的递归数据类型 Expr。

提示：查看图片数据类型，了解非正式描述如何映射到数据声明。

4 编写一个函数，评估：Expr -> Int，评估整数算术表达式。

如果您对 3 和 4 有正确的答案，您应该可以尝试类似 REPL 中的以下内容：

```
*ex_4_1> 评估 (Mult (Val 10) (Add (Val 6) (Val 3)))
90 : 国际
```

5 编写一个函数 maxMaybe : Ord a => Maybe a -> Maybe a，返回两个输入中较大的一个，或者如果两个输入都为 Nothing，则返回 Nothing。例如：

```
*ex_4_1> maxMaybe (Just 4) (Just 5)
Just 5 : 也许是整数
```

```
*ex_4_1> maxMaybe (Just 4) 没有
Just 4 : 也许是整数
```

6 编写一个函数 maximumTriangle : Picture -> Maybe Double，返回图片中最大三角形的面积，如果没有三角形，则返回 Nothing。

例如，您可以定义以下图片：

```
testPic1 : 图片
testPic1 = 组合 (原始 (三角形 2 3) )
               (原始 (三角形 2 4) )
```

```
testPic2 : 图片
testPic2 = 组合 (原始 (矩形 1 3) )
               (原始 (第 4 圈) )
```

然后，在REPL中测试最大三角形，如下所示：

```
*ex_4_1> 最大三角形 testPic1 Just 4.0 : 也许是 Double
```

```
*ex_4_1> 最大三角形 testPic2
什么都没有 : 也许是双倍的
```

4.2 定义依赖数据类型

依赖数据类型是根据其他值计算得出的类型。您已经看到了依赖类型Vect，其中确切的类型是根据向量的长度计算得出的：

`Vect : Nat -> 类型 -> 类型`

换句话说，Vect的类型取决于它的长度。这为我们提供了额外的类型精确度，我们使用它来通过类型、定义、改进的过程来帮助指导我们的编程。在本节中，您将看到如何定义依赖类型，例如Vect。这个想法的核心是，因为类型和表达式之间没有语法上的区别，所以可以从任何表达式计算类型。

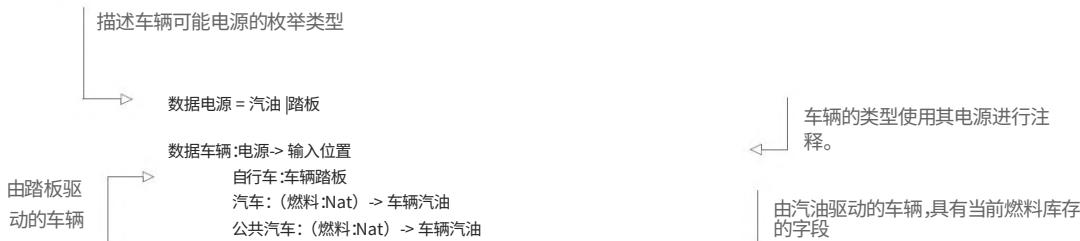
我们将从一个简单的示例开始来说明它是如何工作的，定义一个类型来表示车辆及其属性，具体取决于它们的电源，您将看到如何使用它来将函数的有效输入限制为那些有道理。然后您将看到Vect本身是如何定义的，以及一些有用的操作。

4.2.1 第一个例子：按电源分类车辆

依赖类型允许您通过向类型构造函数添加更多参数来提供有关类型的数据构造函数的更精确信息。例如，您可能有一个数据类型来表示车辆（例如，自行车、汽车和公共汽车），但某些操作对类型中的所有值都没有意义（例如，给自行车加油是行不通的，因为没有油箱）。因此，我们将车辆分为脚踏车和汽油车，并用类型表示。

下面的清单显示了如何在 Idris 中表达这一点。

清单 4.10 定义车辆的依赖类型，其电源在类型 (vehicle.idr)



您可以编写适用于所有车辆的函数，方法是使用类型变量代表电源。例如，所有车辆都有多个轮子。另一方面，并非所有车辆都携带燃料，因此只有为类型表明它由汽油驱动的车辆加油才有意义。下面的清单说明了这两个概念。

定义依赖数据类型

清单 4.11 读取和更新Vehicle的属性

```

轮子:车辆功率-> Nat
轮子自行车 = 2
车轮 (汽车燃料)= 4
车轮 (公共汽车燃料)= 4

加油:车用汽油 -> 车用汽油
refuel (汽车燃料)=汽车100
加油 (巴士燃料)= 巴士 200

```

使用类型变量 power,因为此
函数适用于所有可能的车辆类
型。

加油只对携带燃料的车辆有意义,因此将输入
和输出类型限制为车辆汽油。

断言输入是不可能的

如果您尝试添加一个为自行车加油的案例,Idris 将报告类型错误,因为
输入类型仅限于以汽油为动力的车辆,如果您使用交互式
工具,在使用 Ctrl-Alt-C 拆分案例后,Idris 甚至不会为 Bicycle 提供案例。尽管如此,它有时可以帮助提高可读
性,明确表明您知道
自行车情况是不可能的。你可以这样写:

```

加油:车用汽油 -> 车用汽油
refuel (汽车燃料)=汽车100
加油 (巴士燃料)= 巴士 200
加油自行车不可能

```

如果您这样做,Idris 将检查您标记为不可能的案例是否会
产生类型错误。

同样,如果您断言一个案例是不可行的,但 Idris 认为它是有效的,它会报告
一个错误:

```

加油:车用汽油 -> 车用汽油
refuel (汽车燃料)=汽车100
加油 (公共汽车燃料)不可能

```

在这里,伊德里斯将报告以下内容:

vehicle.idr:15:8:refuel (Bus fuel) 是一个有效的案例

通常,您应该通过给出类型构造函数和
数据构造函数。这为构造函数可以采用的形式提供了很大的灵活性。在这里,它允许您定义数
据构造函数的类型
任何人都可以接受不同的论点。您可以编写适用于所有车辆 (如车轮) 的函数,也可以编写仅适
用于某些车辆子集的函数 (如加油)。

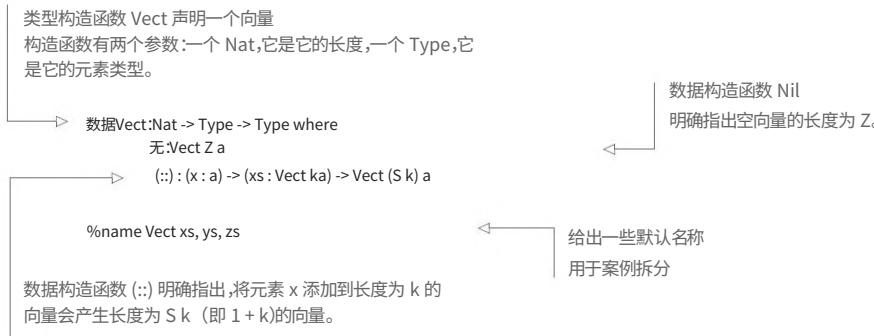
定义类型族对于车辆,您实际上定义了两种类型
在一份声明中 (特别是车辆踏板和车辆汽油)。
因此,像Vehicle这样的依赖数据类型有时被称为
类型系列,因为您同时定义了多个相关类型
时间。电源是车辆家族的一个指标。指数告诉你
你到底是哪种车辆类型。

4.2.2 定义向量

在第3章中,我们研究了如何使用类型中的长度信息来帮助推动向量函数的开发。在本节中,我们将看看Vect如何被定义,以及对它的一些操作。

它在Data.Vect模块中定义,如清单4.12所示。类型构造函数Vect将长度和元素类型作为参数,因此当您定义数据构造函数,你在它们的类型中明确声明它们的长度是多少。

清单4.12 定义向量 (Vect.idr)



Data.Vect库包含Vect上的几个实用函数,包括连接、通过值在向量中的位置查找值以及各种高阶函数,例如map。但是,我们将使用我们自己的定义,而不是导入它

Vect并尝试手动编写一些函数。首先,创建一个Vect.idr文件,其中仅包含清单4.12中Vect的定义。

因为Vect在其类型中明确包含长度,所以任何使用一些Vect的实例将在其类型中明确描述其长度属性。例如,如果你在Vect上定义了一个append函数,它的类型将表示输入和输出是相关的:

附加:Vect n elem -> Vect m elem -> Vect (n + m) elem

TYPE-LEVEL EXPRESSIONS这里返回类型中的表达式 $n + m$ 是一个Nat类型的普通表达式,使用普通的+运算符。因为Vect的第一个参数是Nat类型,你应该期望能够使用任何Nat类型的表达式。请记住,类型是一等的,所以类型和表达式都是同一种语言的一部分。

与往常一样,首先编写类型后,您可以在第一个论点。您可以交互地执行此操作,如下所示:

1定义 首先创建一个骨架定义,然后在第一个参数xs:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ?append_rhs_1
append (x :: xs) ys = ?append_rhs_2
```

2 Refine Idris 在类型中有足够的信息来通过对每个孔的表达式搜索来完成此定义,结果如下:

```
append : Vect n elem -> Vect m elem -> Vect (n + m) elem
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

术语:参数和索引 Vect 定义了一系列类型,我们说一个 Vect 由其长度索引并由元素类型参数化。参数和索引的区别如下:

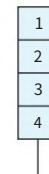
一个参数在整个结构中是不变的。在这种情况下,向量的每个元素都具有相同的类型。

索引可能会在结构中发生变化。在这种情况下,每个子向量都有一个不同的长度。

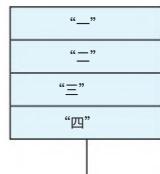
在查看函数的类型时,这种区别最有用:您可以确定参数的特定值不会在函数的定义中发挥作用。然而,正如您在第 3 章中通过查看长度索引定义向量的长度时以及在定义 createEmpties 以构建空向量的向量时所看到的那样,索引可能是这样的。

另一个常见的向量操作是zip,它将两个向量中的对应元素配对,如图 4.5 所示。

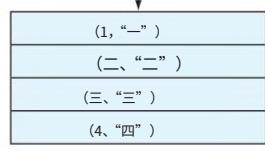
Vect 4 Nat



Vect 4 字符串



zip [1,2,3,4] [“一”、“二”、“三”、“四”]



Vect 4 (自然,字符串)

图 4.5 使用 zip 对 [1,2,3,4] 和 [one , two , three , four] 的对应元素进行配对

`zip`这个名称旨在暗示拉链的工作原理,将包或夹克的两侧连接在一起。因为每个输入`Vect`的长度都在类型中,所以你需要考虑输入和输出的长度将如何对应。一个合理的选择是要求两个输入的长度相同:

```
zip : Vect na -> Vect nb -> Vect n (a, b)
```

为`Vect`提供更精确的类型,捕获类型中的长度,意味着您需要提前确定`zip`输入的长度如何关联并在类型中表达此决定。此外,这意味着您可以放心地假设`zip`只会被等长列表调用,因为如果违反此假设,Idris 将报告类型错误。

您可以像往常一样一步一步地定义`zip`: 1 定义

同样,您可以通过在第一个参数上拆分大小写来开始定义它:

```
zip : Vect na -> Vect nb -> Vect n (a, b) zip [] ys = ?zip_rhs_1 zip (x :: xs) ys = ?
zip_rhs_2
```

2 Refine 您可以使用表达式搜索填充`?zip_rhs_1`,因为唯一类型良好的结果是一个空向量:

```
zip : Vect na -> Vect nb -> Vect n (a, b) zip [] ys = [] zip (x :: xs) ys = ?zip_rhs_2
```

3 Refine 对于第二种情况,`?zip_rhs_2`,查看孔的类型

看看这是否为您提供了有关该做什么的更多信息:

```
b : 类型 a : 类型
```

```
x:a  
k : 晚上  
xs : 你是 Vect  
ys : Vect (S k) b
```

```
zip_rhs_2 : Vect (S k) (a, b)
```

请注意, `ys`的长度为`S k`,这意味着必须至少有一个元素。

如果您在`ys`上进行大小写拆分, Idris 不会为您提供空列表的模式,因为它不是一个类型良好的值:

```
zip : Vect na -> Vect nb -> Vect n (a, b) zip [] ys = [] zip (x :: xs) (y :: ys) = ?zip_rhs_1
```

案例拆分后,伊德里斯又挖了一个新坑,我们来看看局部变量的类型:

```
b : 类型 a : 类型
```

```
x:a  
k : 晚上
```

```

xs : 你是 Vect
y:b ys :
Vect kb
-----
zip_rhs_1 : Vect (S k) (a, b)

```

同样,有足够的信息来完成表达式搜索的定义:

```

zip : Vect na -> Vect nb -> Vect n (a, b) zip [] ys = [] zip (x :: xs) (y :: ys) = (x, y) :: zip
xs ys

```

Idris 注意到它需要构建一个长度为 $S k$ 的向量,它可以通过递归调用创建适当的长度为 k 的向量,并且它可以通过配对 x 和 y 来创建适当的第一个元素。

解构表达式搜索要了解表达式搜索所做的工作,删除部分结果并将其替换为一个孔,以查看当时正在使用什么类型的表达式搜索可能是有益的。例如,您可以删除 (x, y) :

```

zip : Vect na -> Vect nb -> Vect n (a, b) zip [] ys = [] zip (x :: xs) (y :: ys) = ?element ::
zip xs ys

```

然后,检查 $?element$ 孔的类型,您会看到此时 Idris 正在寻找一对 a 和 b :

```
b : 类型 a : 类型
```

```

x:a
k : 晚上
xs : 你是 Vect
y:b ys :
Vect kb
-----

```

元素: (a,b)

此时制作一对 a 和 b 的唯一方法是使用 x 和 y ,所以这就是 Idris 用来构造这对的方法。

4.2.3 使用 Fin 索引有界数的向量

因为 $Vects$ 将它们的长度作为其类型的一部分,所以类型检查器具有额外的知识,可用于检查操作是否已正确实现和使用。一个例子是,如果您希望通过向量中的位置来查找 $Vect$ 中的元素,您可以在编译时知道该位置在程序运行时不能越界。

在Data.Vect 中定义的index函数是一个边界安全的查找函数,它的类型保证它永远不会访问向量边界之外的位置:

索引 : Fin n -> Vect na -> a

第一个参数,类型为Fin n,是一个无符号数,其上界不包含n。Fin这个名字暗示这个数字是有界的。因此,例如,当您按位置查找元素时,您可以使用向量范围内的数字:

```
Idris> :module Data.Vect *Data/Vect>
Vect.index 3 [1,2,3,4,5]
```

在REPL中导入模块在元素查找示例中,您使用:module命令在REPL中导入Data.Vect ,以访问索引函数。Prelude中有几个称为index的函数用于索引不同的类似列表的结构,因此您必须在此处使用Vect.index 明确消除歧义。

但是如果你尝试使用超出范围的数字,你会得到一个类型错误:

```
*Data/Vect> Vect.index 7 [1,2,3,4,5] (输入) 1:14:检查参数 prf 到
函数 Data.Fin.fromInteger 时:
    当使用 7 作为 Fin 5 的文字时,7 不严格小于 5
```

整数文字表示法与 Nat 一样,您可以对Fin使用整数文字,前提是编译器可以确保文字在类型中规定的范围内。

如果您正在读取一个数字作为将用于索引Vect的用户输入,则该数字并不总是在Vect 的范围内。在实践中,您经常需要将任意大小的Integer转换为有界Fin。

导入Data.Vect可让您访问integerToFin函数,该函数将Integer转换为具有一定界限的Fin ,前提是Integer在界限内。它有以下类型:

```
integerToFin : 整数 -> (n : Nat) -> Maybe (Fin n)
```

第一个参数是要转换的整数,第二个参数是Fin的上界。请记住, Fin upper 类型,对于upper 的某些值,表示直到但不包括upper 的数字,因此5不是有效的Fin 5,但4是。这里有几个例子:

```
*数据/向量> integerToFin 2 5
只是 (FS (FS FZ)) :也许 (Fin 5)
```

```
*数据/向量> integerToFin 6 5
什么都没有:也许 (Fin 5)
```

FIN CONSTRUCTORS FZ和FS是Fin 的构造子,对应于Z和S作为Nat 的构造子。通常,您可以使用数字文字,如Nat。

使用integerToFin,您可以编写一个tryIndex函数,通过Integer索引在Vect中查找值,在类型中使用Maybe来捕获结果可能超出范围的可能性。首先创建一个导入Data.Vect 的 TryIndex.idr 文件。然后,按照以下步骤操作:

1种类型 和以往一样,首先给出一个类型:

```
tryIndex : 整数 -> Vect na -> 也许一个
```

请注意,此类型在输入Integer和Vect的长度之间没有任何关系。

2定义 您可以编写定义,使用integerToFin检查输入是否在范围内:

```
tryIndex : Integer -> Vect na -> 也许是 tryIndex {n} i xs = case integerToFin
in of case_val => ?tryIndex_rhs
```

请注意,您需要将n带入范围,以便您可以将其传递给integerToFin作为Fin的所需边界。

3定义 现在,通过case_val上的案例拆分来定义函数。如果integerToFin返回Nothing,则输入超出范围,因此您返回Nothing:

```
tryIndex : Integer -> Vect na -> Maybe a tryIndex {n} i xs = case integerToFin
in of
    没有 => 没有 只是 idx => ?
    tryIndex_rhs_2
```

4类型 如果你检查tryIndex_rhs_2 的类型,你会看到你现在有一个Fin n和Vect n a,因此您可以安全地使用索引:

```
n : 晚上
idx : 镜
a : 类型 i : 整数
xs : Vect na
```

tryIndex_rhs_2 : 也许是

最终结果如下:

```
tryIndex : Integer -> Vect na -> Maybe a tryIndex {n} i xs = case integerToFin
in of
    没有=>没有
    只是 idx => 只是 (索引 idx xs)
```

这是依赖类型编程中的一种常见模式,您将在接下来的章节中更频繁地看到这种模式。索引的类型会告诉您何时可以安全调用它,因此如果您有可能不安全的输入,则需要检查。将Integer转换为Fin n后,您就知道该数字必须在界限内,因此您无需再次检查。

练习

- 1 扩展 Vehicle 数据类型,使其支持独轮车和摩托车,以及更新车轮并相应地加油。
- 2 扩展 PowerSource 和 Vehicle 数据类型以支持电动汽车(例如如电车或电动汽车)。
- 3 List 上的 take 函数的类型为 Nat -> List a -> List a。什么是适当的 Vect 上对应的 vectTake 函数的类型?

提示:输入和输出的长度如何关联?它不应该是有效的采用比 Vect 中更多的元素。另外,请记住,您可以拥有类型中的任何表达式。

- 4 实施 vectTake。如果你用正确的类型正确地实现了它,你可以在 REPL 上测试您的答案,如下所示:

```
*ex_4_2> vectTake 3 [1,2,3,4,5,6,7]
[1, 2, 3] : Vect 3 整数
```

如果您尝试使用太多元素,您还应该收到类型错误:

```
*ex_4_2> vectTake 8 [1,2,3,4,5,6,7]
(输入) 1:14 : 检查构造函数 Main...:: 的参数 xs 时
        类型不匹配
                  Vect 0 a1 ([] 类型)
        和
                  Vect (S m) a (预期类型)
```

- 5 编写具有以下类型的 sumEntries 函数:

```
sumEntries : Num a => (pos : Integer) -> Vect to -> Vect to -> Maybe a
```

如果 pos 是,它应该返回每个输入中位置 pos 的条目总和范围内,否则没有。例如:

```
*ex_4_2> sumEntries 2 [1,2,3,4] [5,6,7,8]
Just 10 : 也许是整数
```

```
*ex_4_2> sumEntries 4 [1,2,3,4] [5,6,7,8]
Nothing : 也许是整数
```

提示:你需要调用 integerToFin,但你只需要调用一次。

4.3 交互式数据存储的类型驱动实现

为了将您到目前为止学到的想法付诸实践,现在让我们看一下更大的示例程序,交互式数据存储。在本节中,我们将设置基本的基础设施结构。当您了解有关 Idris 的更多信息时,我们将在第 6 章修改这个程序,以支持键值对,以及用于描述数据形式的模式。

在我们最初的实现中,我们只支持将数据作为字符串存储在内存中,通过数字标识符访问。它将有一个命令提示符并支持以下命令:

add [String] 将字符串添加到文档存储并通过打印
 您可以通过它来引用字符串的标识符。
 get [Identifier] 检索并打印具有给定标识符的字符串，
 假设标识符存在，否则出现错误消息。
 quit 退出程序。

一个简短的会议可能如下：

```
$ ./datastore 命令:添加
Even Old New York
编号 0
命令:添加曾经是新阿姆斯特丹
编号 1
命令:得到 1
曾经是新阿姆斯特丹
命令:得到 2
超出范围
命令:添加他们为什么改变它我不能说
编号 2
命令:得到 2
为什么他们改变它我不能说
命令:退出
```

我们将使用类型系统来保证对数据存储的所有访问都使用有效的标识符，并且我们所有的函数都是完整的，因此我们确保程序不会由于意外输入而中止。

在高层次上，我们将采用的总体方法再次遵循类型、定义、细化的过程：

类型 为数据存储的表示设计一种新的数据类型。在类型驱动的开发中，即使在最高级别，类型也是第一位的。在我们实现数据存储程序的任何部分之前，我们需要知道我们是如何表示和处理数据的。
定义 尽可能多地实现一个主程序，为我们不能立即编写的部分留下漏洞，将这些漏洞提升到顶层函数。
细化 随着我们深入实现并提高对问题的理解，我们将根据需要细化实现和类型。

首先，创建一个 DataStore.idr 文件的大纲，该文件包含一个模块头、一个 Data.Vect 的导入语句和一个空的 main 函数，如下所示。

清单 4.13 数据存储 (DataStore.idr) 的概要实现

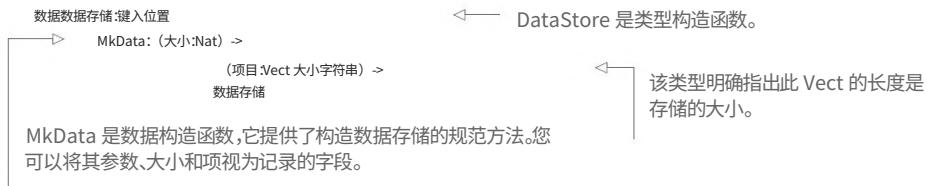
模块主要	
导入 Data.Vect	使用 Vect 存储数据，以便您可以跟踪类型中 存储的大小
main : IO () main = ? main_rhs	main 的初始实现为空

我们将首先定义一个表示存储的类型,然后我们将编写一个读取用户输入并根据用户命令更新存储的主函数。正如我们在实现过程中,我们将根据需要添加新的类型和功能,始终由 Idris 类型检查器指导。

4.3.1 代表店铺

最初,数据存储本身是字符串的集合。我们将从定义一个类型开始对于商店,包括商店的大小(即存储物品的数量),显式地为项目使用Vect,如下面的清单所示。你可以加这个到DataStore.idr,上面main的初始空定义。

清单 4.14 表示数据存储的数据类型 (DataStore.idr)



您可以通过编写模式来访问数据存储的大小和内容
匹配数据存储并提取适当的字段。这些显示在以下列表中。

清单 4.15 投影出数据存储 (DataStore.idr) 的大小和内容

```

    size: DataStore -> Nat
    size (MkData size project) = size

    items : (store : DataStore) -> Vect (size store) String
    project (MkData size project) = project
  
```

右侧注释说明 Vect 类型中投射到商店外的长度由投射到商店外的大小给出。

在这个清单中,项目类型中Vect的长度是由一个函数计算的,尺寸。

RECORDS具有一个构造函数的数据类型,如DataStore,本质上是一个记录其数据的字段。在第 6 章中,您将看到更简洁的语法对于不需要编写显式投影函数的记录,例如大小和项目。

您还需要将新数据项添加到存储中,如清单 4.16 所示。这增加了项目到 store 的末尾,而不是在开头直接使用::。的原因这是我们计划通过整数索引访问项目;如果您在开头添加项目,新项目的索引将始终为零,其他所有内容都将移动沿一处。

交互式数据存储的类型驱动实现

清单 4.16 向数据存储 (DataStore.idr) 添加一个新条目

您可以使用 `_` 在这里,而不是给出大小
明确地,因为 Idris 可以在编译时根据 `addToData` 的类型
计算出新的大小。

```
addToString : 数据存储 -> 字符串 -> 数据存储
addToString (MkData 大小项目) newItem = MkData
    在哪里
    _ (addToString 项)
    addToString : Vect 旧字符串 -> Vect (S old) 字符串
    addToString [] = [newItem]
    addToString (item :: items) = item :: addToString items
```

这种类型表明 `addToString` 总是将
存储的长度增加 1。

在 `where` 块中,函数可以访问
到外部的模式变量
函数,所以你可以在这里使用 `newItem`。

where 块中的交互式编辑

交互式编辑工具在 `where` 块中的工作与在
顶层。例如,尝试从此时开始实现 `addToString`:

```
addToString : 数据存储 -> 字符串 -> 数据存储
addToString (MkData 大小项目) newItem
    = MkData
    在哪里
    _ (addToString 项)
    addToString : Vect 旧字符串 -> Vect (S old) 字符串
```

您可以使用 Ctrl-Alt-A 添加 `addToString` 的定义,并使用表达式搜索
Ctrl-Alt-S 知道 `newItem` 在范围内。

4.3.2 交互维护main中的状态

当您为数据存储实现主要功能时,您需要读取输入
从用户,维护商店本身的状态,并允许用户退出。在里面
上一章,我们使用 Prelude 函数 `repl` 编写了一个简单的交互程序
重复读取输入,在其上运行一个函数,并显示输出:

`repl : 字符串 -> (字符串 -> 字符串) -> IO ()`

不幸的是,这只允许永远重复的简单交互。

对于维护状态的更复杂的程序,Prelude 提供了另一个
函数 `replWith`,它实现了一个携带一些状态的read-eval-print循环。
`:doc` 描述如下:

```
伊德里斯> :doc replWith
Prelude.Interactive.replWith : (state : a) ->
    (提示:字符串) ->
    (onInput : a -> String -> Maybe (String, a)) -> IO ()
```

一个基本的读取-评估-打印循环,维护一个状态
 论据:
 state : a -- 输入状态
 prompt : String -- 要显示的提示
 onInput : a -> String -> Maybe (String, a) -- 函数
 在读取输入时运行,返回一个字符串到输出和一个新的
 状态。如果 repl 应该退出,则返回 Nothing

在循环的每次迭代中,它都会调用onInput参数,该参数本身采用
 两个论点:

当前状态,一些泛型类型

在提示符下输入的字符串

onInput函数应该返回的值是Maybe (String, a) 类型,意思是
 它可以是以下形式之一:

什么都没有,如果它想退出循环

只是 (output, newState),如果它想打印一些输出并更新
 state 到newState用于下一次迭代

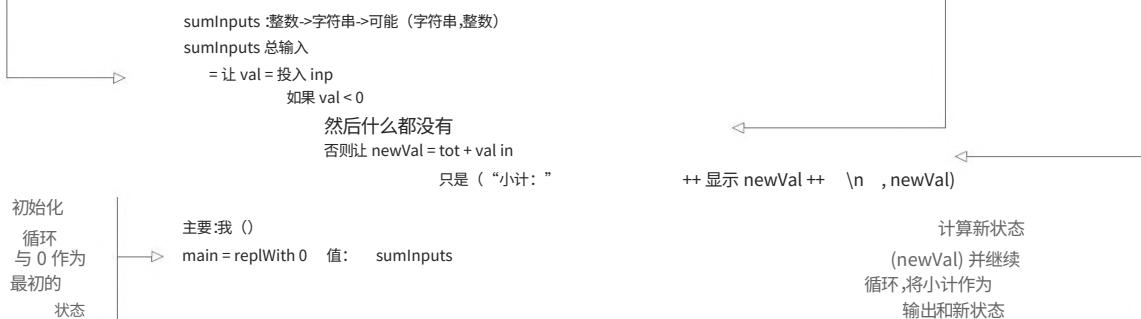
下一个清单显示了一个简单的例子:一个交互式程序

从控制台读取一个整数并显示总和的总和
 输入。如果它读取一个负值,它将退出。

清单 4.17 对输入值求和直到读取到负值的交互式程序 (SumInputs.idr)

将输入字符串转换为整数。如果输入不是有效数字,则默认值为 0。
 Idris 知道 val 必须是 Integer,因为它稍后会在只有 Integer 类型
 正确的上下文中使用。

收到负输入,所以
 返回
 无,退出循环



您可以使用replWith来优化数据存储中的主要功能。在这个阶段你有

- 数据存储 (DataStore) 的类型
- 一种访问商店中物品 (物品) 的方式
- 一种使用新项目更新商店的方法 (addToString)

调用replWith时,需要将初始数据、提示和函数传递给处理输入作为参数。您可以传递一个初始化的空数据存储和一个提示字符串,但您还没有处理用户输入的函数。尽管如此,您可以将main的定义细化为以下内容,为输入处理功能留下一个洞:

```
主要:我 ()
main = replWith (MkData
    _ [])) 命令: ?processInput
```

Lifting processInput显示了您必须使用的类型:

```
processInput:数据存储->字符串->可能 (字符串,数据存储)
主要:我 ()
main = replWith (MkData
    _ [])) 命令: 进程输入
```

遵循类型驱动的方法,当您使用replWith的应用, Idris 能够为进程输入。

4.3.3 命令:解析用户输入

要处理字符串输入,您必须以某种方式确定输入了哪个命令add、get或quit。而不是处理输入字符串

直接地,定义一个代表可能的新数据类型通常要干净得多命令。这样,您可以将命令的解析与加工。

因此,您将定义一个Command数据类型,它是一个联合类型,表示可能的命令及其参数。将以下定义放入 DataStore.idr
以上流程输入:

```
数据命令 = 添加字符串
    | 获取整数
    | 退出
```

避免使用字符串表示数据用户输入字符串,
但只有一定数量的字符串是有效的输入命令。引入Command类型使命令的表示更加精确

因为只能表示有效的命令。如果用户输入一个字符串无法转换为命令,类型迫使您考虑如何来处理错误。起初,您可以在程序中留下一个漏洞以防出错处理,但最终,制作精确的类型会导致更健壮执行。

您需要将用户输入的字符串转换为命令。输入可能是
然而,无效,因此解析命令的函数类型在其类型中捕获了这种可能性:

解析: (输入:字符串) -> 也许命令

您可以为输入命令编写一个简单的解析器,方法是使用函数搜索输入中的第一个空格,以确定输入的哪一部分是命令,哪一部分是参数。 span函数的工作原理如下:

```
Idris> :t Strings.span
Strings.span : (Char ->
  Bool) -> String -> (String, String)

Idris> span (/=      )  Hello world, here is a string
(Hello , world, here is a string ) : (String, String)
```

第一个参数(/=)是一个返回Bool 的测试。对于不等于空格的任何字符,此测试返回True 。第二个参数是输入字符串, span会将字符串分成两部分:

第一部分是输入字符串的前缀,所有字符都满足
测试。

第二部分是字符串的剩余部分。如果字符串中的所有字符都坐
isfy 测试,这将是空的。

您可以使用 Ctrl-Alt-A 定义 parse ,然后将其定义细化为以下内容:

```
parse : (input : String) -> Maybe Command
parse input = case span (/=
) input of (cmd, args) => ?parseCommand
```

然后,您可以将parseCommand提升到具有适当类型的顶级函数,使用
Ctrl-Alt-L:

```
parseCommand: (cmd:字符串) -> (args:字符串) -> (输入:字符串) ->
也许命令

parse : (input : String) -> Maybe Command
parse input = case span (/=
) input of (cmd, args) => parseCommand cmd args input
```

您不需要输入参数,因为您将单独解析来自cmd和args的输入,尽管 Idris 已添加它,因为输入在范围内。因此,您可以编辑类型:

```
parseCommand : (cmd : String) -> (args: String) -> Maybe 命令

parse : (input : String) -> Maybe Command
parse input = case span (/=
) input of
  (cmd, args) => parseCommand cmd args
```

此外,您可以看到args如果不为空,将有一个前导空格,因为span遇到的第一个不满足测试的字符将是一个空格。您可以使用ltrim删除前导空格:String -> String函数,该函数返回其输入并删除前导空格字符:

```
parseCommand : (cmd : String) -> (args: String) -> Maybe 命令

parse : (input : String) -> Maybe Command
parse input = case span (/=
) input of (cmd, args) => parseCommand cmd (ltrim args)
```

交互式数据存储的类型驱动实现

您现在可以通过检查cmd和args参数来编写parseCommand。你会需要一些 Prelude 函数来完成parseCommand 的定义：

```
unpack : String -> List Char 将字符串转换为字符列表。
isDigit : Char -> Bool 返回一个Char是否是0-9的数字之一。
all : (a -> Bool) -> List a -> Bool 返回列表中的每个条目是否满足一个测试。
```

因此,表达式all isDigit (unpack val)返回字符串val是否完全由数字组成。

[文档和前奏功能](#)一般来说,请记住,您可以
在REPL 中使用:doc或在 Atom 中使用 Ctrl-Alt-D 来检查文档
任何名称,无论它们是类型构造函数、数据构造函数、
或功能。

清单 4.18 展示了解析输入的工作原理。特别是,请注意该模式
匹配很一般;只要模式由构造类型的原始方式 (数据构造函数和原始值)组成,它们就是有效的。低于分数线是匹配任何内容的
模式。

清单 4.18 将命令和参数字符串解析为命令(DataStore.idr)

```
parseCommand : 字符串 -> 字符串 -> 也许命令
parseCommand add str = Just (Add str)
parseCommand get val = case all isDigit (unpack val) of
    错误 => 没有
    True => Just (Get (cast val))
parseCommand 退出      = 刚退出
解析命令 -- = 没有
此模式匹配
第一个参数是“add”的应用程序
序。您添加到商店的字符串将由整个第二个参数 str 组成。
这匹配任何输入。图案
从上到下匹配工作;
如果没有之前的模式
已匹配,则输入为
无效,所以没有命令。
```

此模式匹配
第一个应用程序
论据是“得到”。解析是
如果第二个参数有效
完全由数字组成。

现在您可以将字符串解析为命令,您可以在以下方面取得更多进展
processInput,调用解析。如果失败,您将显示一条错误消息并离开
按原样存储。否则,您将添加一个用于处理命令的孔:

```
processInput:数据存储->字符串->可能 (字符串,数据存储)
processInput 存储输入
= case parse inp of
    Nothing => Just ( Invalid command\n , store)
    只是 cmd => ?processCommand
```

4.3.4 处理命令

继续执行processInput的一种方法是在cmd上进行大小写拆分并直接处理命令：

```
processInput : DataStore -> String -> Maybe (String, DataStore) processInput store inp
= case parse inp of
    Nothing => Just ( Invalid command\n ,store)
    Just (Add item) => ?processCommand_1 Just (Get pos) => ?
    processCommand_2 Just Quit => ?processCommand_3
```

在添加项目和退出的情况下，您可以通过填充更简单的漏洞来改进实现。下一个清单显示了如何改进这些漏洞，暂时离开进程Command_2。

清单 4.19 处理输入Add item和Quit (DataStore.idr)

```
processInput : DataStore -> String -> Maybe (String, DataStore) processInput store inp = case parse inp of
```

```
Nothing => Just ( Invalid command\n ,store)
只是 (添加项目)->
    只是 (“身份证” ++ show (size store) ++ \n ,addToStore store item)
Just (Get pos) => ?processCommand_2 Just Quit => Nothing
```

返回一个字符串，它给出了添加此项的位置以及使用
addToStore 更新的商店

Quit 命令退出，所以返
回 Nothing

检查 Idris 生成的孔的类型始终是一个好主意，以了解您有哪些可用的变量、它们的类型以及您需要构建的类型。对于?processCommand_2，你有这个：

```
位置:整数
存储:数据存储
输入:字符串
```

```
-----
```

```
processCommand_2:Maybe (字符串,数据存储)
```

这将比其他情况稍微多一些。您需要执行以下操作：

从商店获取物品。

确保位置pos在范围内。如果是，则从指定的pos中提取item

并与store一起返回
本身。

type-define-refine 过程鼓励您逐步编写定义的一部分，不断进行类型检查，并不断检查孔的类型。

因为填充?processCommand_2孔涉及一些细节，
我们将它重命名为?getEntry并在实现之前将其提升为顶级函数
一步步：

1类型 将 getEntry提升到新的顶级函数：

```
getEntry : (pos : Integer) -> (store : DataStore) -> (input : String) ->
    也许 (字符串,数据存储)
```

2定义- 创建新的骨架定义：

```
getEntry pos 存储输入 = ?getEntry_rhs
```

3定义 你需要商店里的物品,所以定义一个新的局部变量
这些：

```
getEntry pos store input = let store_items = items store in
    ?getEntry_rhs
```

检查getEntry_rhs的类型会告诉您store_items 的类型：

```
位置:整数
存储:数据存储
输入:字符串
store_items : Vect (大小存储)字符串
-----
getEntry_rhs : 也许 (字符串,数据存储)
```

4 Refine 要从Vect 中检索条目,您可以像以前一样使用索引
经常看到：

索引 : Fin n -> Vect na -> a

要从类型为Vect (大小存储)的store_items 中提取条目，
你需要一个鳍 (大小商店)。不幸的是,您在
时刻是一个整数。使用integerToFin,如第 4.2.34.2.3 节所述，
您可以细化定义。如果integerToFin返回Nothing,则输入为
出界。

```
getEntry : (pos : Integer) -> (store : DataStore) -> (input : String) ->
    也许 (字符串,数据存储)
getEntry pos 存储输入
= 让 store_items = 物品存放在
    case integerToFin pos (size store) of
        什么都没有 => 只是 ( “超出范围\n” ,存储)
        只是 id => ?getEntry_rhs_2
```

5 Refine 如果你现在检查getEntry_rhs_2的类型,你会发现你有
您需要的Fin (大小商店)：

```
存储:数据存储
id : 鳍 (大小商店)
位置:整数
输入:字符串
store_items : Vect (大小存储)字符串
-----
```

```
getEntry_rhs_2 :也许 (字符串,数据存储)
```

您现在可以细化为完整的定义：

```
getEntry : (pos : Integer) -> (store : DataStore) -> (input : String)
        也许 (字符串,数据存储)
getEntry pos 存储输入
= let store_items = items store in case integerToFin pos (size
    store) of
    什么都没有 => 只是 ( “超出范围\n” ,存储)
    Just id => Just (index id store_items ++ \n ,store)
```

6 Refine 作为最后的细化,请注意从未使用过输入,因此您可以删除此参数。不要忘记将它从 processInput 中的 getEntry 应用程序中删除。

```
getEntry : (pos : Integer) -> (store : DataStore) ->
        也许 (字符串,数据存储)
getEntry pos store = let
    store_items = items store in case integerToFin pos (size store)
    什么都没有 => 只是 ( “超出范围\n” ,存储)
    Just id => Just (index id store_items ++ \n ,store)
```

通过将 Vect 用于存储,并将其大小作为类型的一部分,类型系统可以确保通过索引对存储的任何访问都在界限内,因为您必须证明索引与 Vect 的长度。

作为参考,下表给出了数据存储的完整实现,以及我们刚刚完成的所有功能。

清单 4.20 一个简单数据存储的完整实现 (DataStore.idr)

模块主要

导入数据.Vect

数据数据存储 :键入位置

```
MkData : (size : Nat) -> (items : Vect size String) -> DataStore
```

```
size : DataStore -> Nat size (MkData size
```

```
items ) = size
```

```
items : (store : DataStore) -> Vect (size store) String items (MkData size items ) = items
```

```
addToStore : DataStore -> String -> DataStore addToStore (MkData size store)
```

```
newitem = MkData
```

在哪里

```
addToData : Vect oldsize String -> Vect (S oldsize) String addToData [] = [newitem] addToData (x :: xs) =
x :: addToData xs
```

数据命令 = 添加字符串

```
|获取整数  
|退出
```

交互式数据存储的类型驱动实现

```

parseCommand : String -> String -> Maybe Command parseCommand add str = Just
(Add str) parseCommand get val = case all isDigit (unpack val) of
    错误 => 没有
    True => Just (Get (cast val))
parseCommand 退出 = 退出 parseCommand
-- = 没有

parse : (input : String) -> Maybe Command parse input = case span (/=      )
input of (cmd, args) => parseCommand cmd (ltrim args)

getEntry : (pos : Integer) -> (store : DataStore) ->
    也许 ("字符串", 数据存储)
getEntry pos store = let store_items =
    = items store in case integerToFin pos (size store)

    什么都没有 => 只是 ("超出范围\n" , 存储)
    Just id => Just (index id (items store) ++ \n , store)

processInput : DataStore -> String -> Maybe (String, DataStore) processInput 存储输入
    = 案例解析输入
    Nothing => Just ( Invalid command\n , store)
    只是 (添加项目) =>
        只是 ("身份证" ++ show (size store) ++ \n , addToStore store item)
    Just (Get pos) => getEntry pos store
    只是 退出 => 没有

main : IO () main =
replWith (MkData
    _ [])) 命令： 进程输入

```

练习



- 1 添加一个size命令，显示商店中的条目数。
- 2 添加一个搜索命令，显示商店中包含给定的所有条目子串。
提示：使用Strings.isInfixOf。
- 3 扩展搜索以打印每个结果的位置以及字符串。

您可以在REPL上测试您的解决方案，如下所示：

```

*ex_4_3> 执行
命令:添加采煤机
编号 0
命令:添加米尔本
编号 1
命令:添加白色
编号 2
命令:大小
3
命令:搜索 Mil
1:米尔本

```

4.4 总结

数据类型是根据类型构造函数和数据构造函数定义的。

枚举类型是通过列出该类型的数据构造函数来定义的。

联合类型是通过列出该类型的数据构造函数来定义的,每个构造函数
可能带有附加信息。

泛型类型比其他类型参数化。在泛型类型定义中
化,变量代替具体类型。

依赖类型可以在任何其他值上建立索引。

使用依赖类型,您可以在同一个声明中将较大的类型族 (例如车辆)划分为较小的子组 (例
如汽油驱动的车辆和踏板驱动的车辆)。

依赖类型允许在编译时保证安全检查,例如

保证所有向量访问都在向量的范围内。

您可以以类型驱动的风格编写更大的程序,创建新的数据类型
在适当的地方帮助描述系统的组件。

可以使用replWith函数编写涉及状态的交互式程序。

交互式程序:输入和 输出处理

本章涵盖

编写交互式控制台程序
区分评估和执行验证
用户对依赖类型函数的输入

Idris 是一种纯语言,这意味着函数没有副作用,例如更新全局变量、抛出异常或执行控制台输入或输出。但实际上,当我们将功能组合在一起以制作完整的程序时,我们需要以某种方式与用户进行交互。

在前面的章节中,我们使用repl和replWith函数编写了简单的循环交互式程序,我们可以编译和执行这些程序,而不必过多担心它们的工作原理。然而,对于除了最简单的程序之外的所有程序,这种方法是非常有限的。在本章中,您将了解 Idris 中交互式程序是如何工作的。您将看到如何处理和验证用户输入,以及如何编写使用依赖类型的交互式程序。

尽管 Idris 是一种纯语言,但允许我们用 Idris 编写交互式程序的关键思想是我们区分评估和执行。我们使用通用类型 IO 编写交互式程序,它描述了随后由 Idris 运行时系统执行的操作序列。

5.1 使用 IO 进行交互式编程

尽管您不能编写直接与用户交互的函数,但您可以编写描述交互序列的函数。一旦你有了一系列交互的描述,你就可以将它传递给 Idris 运行时环境,它会执行这些动作。

Prelude 提供了一个泛型类型, IO, 它允许您描述交互式程序并返回值的克数:

```
伊德里斯> :文档 IO
IO : (res : Type) -> Type 交互式程序, 描述 I/O
    动作并返回一个值。
```

论据:

```
res : Type -- 程序的结果类型
```

因此,您可以区分描述与用户交互的函数和直接返回值的函数的类型。

例如,考虑函数类型 String -> Int 和 String -> IO Int 之间的区别:

String -> Int 是一个函数的类型,它接受一个 String 作为输入并返回一个 Int 而没有任何用户交互或副作用。String -> IO Int 是一个函数的类型,它接受一个字符串作为输入并返回一个生成一个 Int 的交互式程序的描述。

这些是具有这些类型的函数的几个示例:

```
length : String -> Int, 在 Prelude 中定义, 返回长度
    的输入字符串。
readAndGetLength : String -> IO Int, 返回交互式操作的描述, 将输入字符串显示为提示, 从控制
    台读取另一个字符串, 然后返回该字符串的长度。
```

正如您在前面的章节中所看到的, Idris 应用程序的入口点是 main : IO (), 我们使用它使用 repl 和 replWith 函数编写简单的交互式循环, 而不必过多担心这些函数的细节。功能起作用。

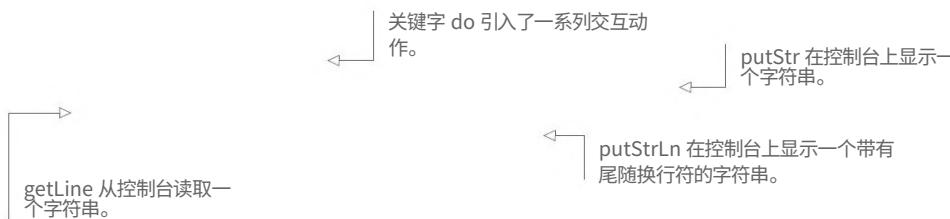
现在是时候更详细地了解 IO, 并学习如何编写更复杂的交互式程序了。

清单 5.1 显示了一个交互式程序的示例, 它返回操作的描述以显示提示、读取用户名, 然后显示问候语。

我们将在本节中介绍语法的细节, 但现在, 请注意 main 的类型: IO ()。这种类型声明 main 不接受任何输入并返回生成空元组的交互操作的描述。

清单 5.1 一个简单的交互式程序,读取用户名并显示问候语

荷兰国际集团 (Hello.idr)



HASKELL 和 IDRIS 中的 IO如果你已经熟悉 Haskell,你会发现
在 Idris 中使用IO进行编程与在 Haskell 中编写交互式程序非常相似。如果你理解了清单 5.1,
你可以安全地进入第 5.3 节,在那里你将看到如何验证用户输入和处理错误

交互式 Idris 程序。您可能还想查看第 5.2.2 节,其中
讨论模式匹配绑定。

在本节中,您将学习如何在 Idris 中使用IO编写这样的交互式程序
并探索评估表达式和执行程序之间的区别,
这使我们能够在不影响纯度的情况下编写交互式程序。

5.1.1 评估和执行交互式程序

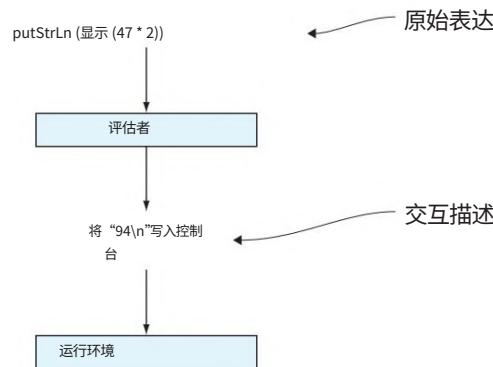
返回IO类型的函数仍然被认为是纯函数,因为它们只是描述
互动动作。例如, putStrLn 函数在 Prelude 和
将输出给定字符串和换行符的操作返回到控制台:

当您在REPL 中输入表达式时, Idris 会计算该表达式。如果说
表达式是对互动动作的描述,然后执行这些动作
需要一个额外的步骤,执行。

图 5.1 说明了当表达式 putStrLn (show (47 * 2)) 时会发生什么
被执行。首先,Idris 计算出互动动作是在屏幕上显示 “94\n”
控制台 (即评估器计算要显示的确切字符串),以及
然后它将该操作传递给运行时环境。

您可以通过计算表达式 putStrLn (show (47 * 2)) 来查看发生了什么
在REPL。这仅显示了运行时环境可以执行的操作的描述。这里不需要仔细查看结果的确切形式,

但你至少可以看到评估产生了一个IO () 类型的表达式:



如果要执行结果操作,则需要将此描述传递给 Idris 运行时环境。您可以使用REPL中的:exec命令实现此目的:

一般来说,给定IO ()类型的表达式的:exec命令可以理解为执行以下操作:

1评估表达式,生成要执行的交互操作的描述。在图 5.1 中,对表达式求值会生成一个动作描述:将字符串 “94\\n”写入控制台。

2将生成的操作传递给执行它们的运行时环境。在图 5.1 中,执行操作会将字符串 “94\\n”写入控制台。

也可以使用REPL中的:c命令编译为独立的可执行文件,该命令将可执行文件名称作为参数并生成一个执行main 中描述的操作的程序。例如,让我们回到清单 5.1,在此重复:

如果你将它保存在一个文件 Hello.idr 中,并在REPL 中加载它,你可以生成一个可执行文件,如下所示:

这会生成一个名为hello (或在 Windows 系统上为hello.exe)的可执行文件,该文件可以直接从 shell 运行。例如:

在这里， main不仅描述了一个动作，而且描述了一系列动作。通常，在交互式程序中，您需要能够对操作进行排序并让命令对用户输入做出反应。因此，Idris 提供了将较小的交互式程序组合成较大程序的工具。我们将首先了解如何使用 Prelude 函数($>=$) 来执行此操作，然后我们将了解我们之前在清单 5.1 中使用的高级语法，即do表示法。

5.1.2 动作和排序: $>=$ 运算符

在清单 5.1 的简短示例中，我们使用了三个IO操作： putStr : String -> IO ()，

这是一个在控制台上显示字符串的操作 putStrLn : String -> IO ()，这是一个操作在控制台上显示一个字符串，后跟一个换行符

getLine : IO String，这是一个从控制台读取String的动作

为了编写逼真的程序，您需要做的不仅仅是执行操作。

您需要能够对操作进行排序，并且您需要能够处理这些操作的结果。

比如说，你想从控制台读取一个字符串，然后输出它的长度。 Prelude 中有一个长度函数，类型为String -> Int，可用于计算字符串的长度，但getLine返回的是IO String 类型的值，而不是String。 IO String类型是对产生String 的操作的描述，而不是String本身。

换句话说，您可以在执行函数产生的操作时访问从控制台读取的字符串，但在评估函数时您还不能访问它。没有 IO String -> String 类型的函数。如果存在这样的函数，则意味着可以知道从控制台读取了哪个字符串，而无需实际从控制台读取任何字符串！

Prelude 提供了一个名为 $>=$ 的函数（旨在用作中缀运算符），它允许对IO操作进行排序，将一个操作的结果作为输入提供给下一个操作。它有以下类型：

$(>=) : I a -> (a -> I b) -> I b$

图 5.2 显示了 $>=$ 对两个动作进行排序的示例应用程序，将第一个动作的输出getLine作为输入输入到第二个动作 putStrLn。因为getLine返回一个IO String 类型的值，Idris 期望putStrLn 将String作为其参数。

您可以使用:exec命令在REPL中执行任何类型为IO ()的操作序列，因此您可以执行图 5.2 中的操作，如下所示：

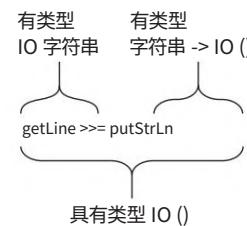


图 5.2 读取字符串并使用 $>=$ 操作符回显其内容的交互式操作。指示了每个子表达式的类型。

7 Refine 最后,使用show完成定义,将len转换为String,并putStrLn给出显示结果的操作:

```
打印长度:IO ()  
printLength = getLine >>= \input => 让 len = 长度输入  
                                putStrLn (显示长度)
```

你可以在REPL中用:exec printLength尝试这个定义,告诉Idris您希望执行结果操作,而不仅仅是评估表达式:

```
*打印长度> :exec printLength  
测试输入  
10
```

由用户输入
Idris 运行时环境的输出

清单 5.2 显示了完整的定义,并通过显示提示进一步细化。在此定义中的布局也进行了调整,以突出显示操作的顺序。

**清单 5.2 显示提示,读取字符串,然后显示其长度的函数,
使用>>对IO操作进行排序(PrintLength.idr)**

```
打印长度:IO ()  
printLength = putStr    输入字符串:  getLine >>= \input =>      >>= \_ =>  
                           让 len = 长度输入  
                           putStrLn (显示长度)  
  
                           getLine 返回一个 IO 字符串,所以在  
                           定义的其余部分,输入将具有字符串  
                           类型。
```

putStr 返回 IO (),因此 >>= 的
第二个参数需要一个 () 类型的
值作为其输入。下划线表示忽略
此值。

>>= 的类型

如果您在REPL中检查>>=的类型,您将看到一个受约束的泛型类型:

```
伊德里斯> :t (>>=)  
(>>=) : Monad m => ma -> (a -> mb) -> mb
```

在实践中,这意味着该模式比IO更普遍适用,并且您将稍后查看更多内容。现在,只需在此处将类型变量m读取为IO。

原则上,您始终可以使用>>=对IO操作进行排序,提供一个的输出动作作为下一个动作的输入。然而,由此产生的定义有些难看,排序动作特别常见。这就是为什么Idris提供了一种替代语法来对IO操作进行排序,即do表示法。

5.1.3 使用 do 符号进行排序的语法糖

交互式程序本质上是典型的命令式风格,有一个序列的命令,每个命令都会产生一个可以在以后使用的值。>>=函数抓住了这个想法,但由此产生的定义可能难以阅读。

相反,您可以在do块中对IO操作进行排序,这允许您列出执行块时将运行的操作。例如,要打印两件事
继任你会做这样的事情:

```
printTwoThings : IO ()
printTwoThings = do putStrLn "你好"
                  putStrLn "世界"
```

Idris 将do表示法转换为 \geq 的应用。图 5.3 显示了这在最简单的情况下工作,其中要执行一个动作,然后执行更多行动。



图 5.3 在排序操作时使用 \geq 运算符将do表示法转换为表达式。由ty类型的操作生成的值将被忽略,如下划线所示。

动作的结果可以分配给变量。例如,分配结果
使用getLine从控制台读取变量x然后打印结果,你
可以这样写:

```
打印输入:IO ()
printInput = do x <- getLine
               putStrLn x
```

符号 $x <- \text{getLine}$ 表示getLine操作的结果(即
由IO String 类型的动作产生的字符串)将存储在变量x 中,
您可以在do块的其余部分中使用它。使用do表示法对动作进行排序和绑定这样的变量直接转换为 \geq 的应用,如中所
示

图 5.4。

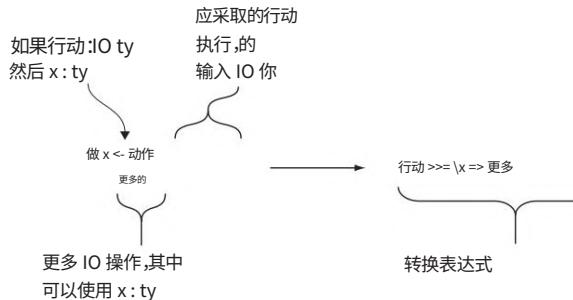


图 5.4 转换do
在绑定变量和排序操作时,使用 \geq 运算符对表达式进行表示法

您还可以在do块中使用let将纯表达式分配给变量。下面的清单显示了如何使用do表示法而不是直接使用`>=`。

**清单 5.3 显示提示、读取字符串并显示其长度的函数
使用do表示法对IO操作进行排序(PrintLength.idr)**

```
打印长度:IO ()  
printLength = do putStrLn "输入字符串:  
    输入<-getLine  
    让 len = 长度输入  
    putStrLn (显示长度)  
  
getLine 的类型为 IO String,因此 input 的类型为 String。  
“长度输入”具有 Nat 类型,因此 len 具有 Nat 类型;它是使用 let 分配的。请注意,在 do 块中,赋值后没有 “in”关键字。
```

let 和 <- 在 do 块中

清单 5.3 中的printLength函数使用两种不同的形式分配给变量:使用 let 和使用`<-`。这两者有一个重要的区别,

这再次依赖于评估和执行之间的区别:

使用`let x = expression`来分配表达式的求值结果到一个变量。

使用`x <- action`将一个动作的执行结果分配给一个变量。

清单 5.3 中, getLine 描述了一个动作,所以它需要被执行并且结果使用`<-`赋值,但长度输入没有,所以使用`let`赋值。

Idris 中的大多数交互式定义都是使用do表示法来控制排序的。在较大的程序中,您还需要响应用户输入和直接程序

流动。在下一节中,我们将了解阅读和回应的各种方法

用户输入,以及如何在交互式程序中实现循环和其他形式的控制流。

练习



1 使用do表示法,编写一个printLonger程序,它读取两个字符串,然后显示较长字符串的长度。

2 使用`>=`而不是do表示法编写相同的程序。

您可以在REPL上测试您的答案,如下所示:

```
*ex_5_1> :exec printLonger  
第一个字符串:短  
第二串:更长  
6
```

5.2 交互程序和控制流程

您已经了解了如何通过对现有操作进行排序来编写基本的交互式程序,例如从控制台读取输入的getLine和在控制台显示文本的putStrLn。但是随着程序变得越来越大,您将需要更多控制:您需要能够验证和响应用户输入,并且您需要表达循环和其他形式的控制。

一般来说,描述交互动作的函数中的控制流的工作方式与纯函数中的控制流完全相同,都是通过模式匹配和递归来实现的。

描述交互动作的函数本身就是纯函数,毕竟只是描述了稍后要执行的动作。

在本节中,我们将研究在交互程序中遇到的一些常见模式:通过组合交互动作的结果产生纯值,对交互动作的结果进行模式匹配,最后将所有内容放在一个交互中带循环的程序。

5.2.1 在交互式定义中产生纯值

除了描述运行时系统执行的操作外,您通常还希望从交互式程序中产生结果。您已经看过getLine,例如:

getLine : IO 字符串

类型IO String表示这是一个描述作为结果生成字符串的操作的函数。

通常,您需要编写一个使用IO操作(如getLine)的函数,然后在返回之前进一步操作其结果。例如,您可能想要编写一个readNumber函数,该函数从控制台读取字符串,如果输入完全由数字组成,则将其转换为Nat。它产生一个Maybe Nat类型的值:

如果输入完全由代表数字i的数字组成,则它产生Just i。例如,在读取字符串“1234”时,它会产生Just 1234。

否则,它不会产生任何东西。

清单 5.4 展示了如何定义readNumber。使用getLine读取输入后,它会检查输入中的每个字符是否都是数字。如果是这样,它将输入转换为Nat,使用Just生成结果;否则,它不会产生任何东西。

清单 5.4 读取和验证数字 (ReadNum.idr)

```
readNumber : IO (也许是 Nat) readNumber =
do
  input <- getLine if all isDigit
    (解压输入)
    then pure (Just (cast input)) else pure Nothing
      ↗ 使用 unpack 将输入转换为
      ↗ List Char, 检查每个输入字符是否为数
      ↗ 字
```

纯函数构造一个产生值的IO动作,没有其他输入或输出效果。

纯函数用于在交互式程序中产生一个值,而在执行时没有任何其他输入或输出效果。其目的是允许交互式程序构造纯值,如其类型所示:

以及:`a -> I a`

要了解`readNumber` 中对`pure`的需求,您可以将`if`的`then`和`else`分支替换为孔,并检查它们的类型:

```
readNumber : IO (也许是 Nat) readNumber =
do
    input <- getLine if all isDigit
        (解压输入)
        然后 ?numberOK
    else ?numberBad
```

如果您查看`?numberOK` 的类型,您会发现您需要一个`IO`类型的值 (可能是 `Nat`) :

```
输入:字符串
-----
numberOK : IO (也许是 Nat)
```

然后,您可以使用 `pure` 细化`?numberOK`和`?numberBad`孔:

```
readNumber : IO (也许是 Nat) readNumber =
do
    input <- getLine if all isDigit
        (unpack input) then pure ?numberOK else pure ?
            numberBad
```

现在,如果您查看`?numberOK` 的类型,您会发现您需要一个`Maybe Nat`类型的值,而不需要`IO`包装器:

```
输入:字符串
-----
numberOK : 也许 Nat
```

纯种

与`>>=`一样,如果您在 REPL 中检查`pure`的类型,您将看到一个受约束的泛型类型,而不是专门使用`IO`的类型:

```
Idris> :t pure pure : 应用
f => a -> fa
```

您可以在此处将 `f` 读取为`IO`。我们将在第 7 章了解`Applicative`和`Monad`的确切含义;您无需了解细节即可编写交互式程序。

您可以在REPL中尝试readNumber ,方法是执行它并将结果传递给println (在边栏中 “使用println 显示值”中进行了描述) :

```
*ReadNum> :exec readNumber >>= println
只读100          ← 用户输入100
只有100          ← 伊德里斯的输出

*ReadNum> :exec readNumber >>= println
没有什么          ← 用户输入错误
没有什么          ← 伊德里斯的输出
```

在第一种情况下,输入有效,所以 Idris 产生结果Just 100。在第二种情况下,输入有非数字字符,所以 Idris 产生Nothing。

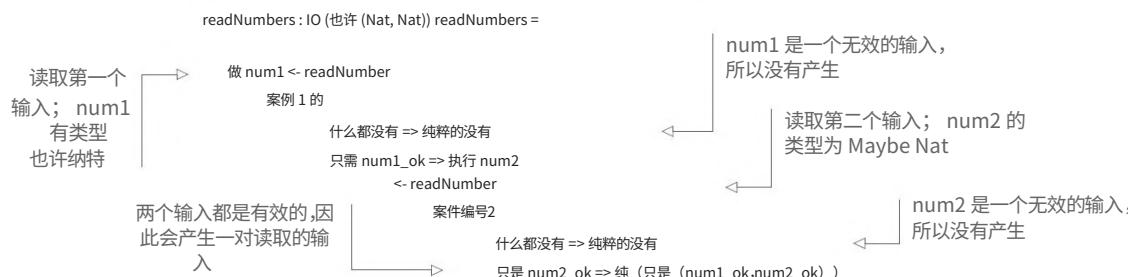
用 println 显示值println是putStrLn 和show的组合,方便直接在控制台显示值。它在 Prelude 中定义:

```
println : 显示 a => a -> IO () println x = putStrLn (显示 x)
```

5.2.2 模式匹配绑定

当一个交互动作产生一个复杂数据类型的值时,例如readNumber,它产生一个Maybe Nat 类型的值,你通常会想要对中间结果进行模式匹配。您可以使用case表达式来执行此操作,但这可能会导致深度嵌套的定义。例如,下面的清单显示了一个函数,它使用readNumber从控制台读取两个数字,如果两个数字都是有效输入,则生成一对数字,否则为Nothing。

清单 5.5 从控制台读取和验证一对数字 (ReadNum.idr)



随着函数变得更长并且有更多的错误条件需要处理,这只会变得更糟。为了解决这个问题,Idris 提供了一些简洁的语法来匹配do表示法中的中间值。首先,我们将看一个简单的示例,从控制台读取一对字符串,然后我们将重新访问readNumbers并了解如何使其更简洁。

您可以编写一个读取两个字符串并生成一对的函数,如下所示,
使用pure组合两个输入：

```
readPair : IO (String, String) readPair = do str1 <- getLine
str2 <- getLine pure (str1, str2)
```

当您使用readPair 生成的结果时,您需要对其进行模式匹配以提取第一个和第二个输入。例如,要使用readPair 读取一对字符串,然后显示两者,您可以这样做：

```
usePair : IO () usePair = do
pair <- readPair case pair of (str1, str2) => putStrLn( You
enter      and
                           ++
str1++           ++ str2)
```

为了使这些程序更简洁,Idris 允许在模式匹配绑定中将模式匹配和赋值组合在一行中。以下程序与前一个程序具有完全相同的行为：

```
usePair : IO () usePair = do
(str1, str2) <- readPair putStrLn( 你输入了
                           ++
str1 ++           " 和 "
                           ++
str2)
```

类似的想法适用于readNumbers。您可以直接对readNumber的结果进行模式匹配以检查其结果的有效性：

```
readNumbers : IO (Maybe (Nat, Nat)) readNumbers = do Just
num1_ok <- readNumber

Just num2_ok <- readNumber pure (Just
(num1_ok, num2_ok))
```

当用户输入有效数字时,这可以按照您的意愿工作：

```
*ReadNum> :exec readNumbers >>= println
20          <---- 用户输入10
只是 (10, 20) <---- 由用户输入
                  <---- 伊德里斯的输出
```

但是,不幸的是,它不处理readNumber产生Nothing 的情况,因此在执行时崩溃：

```
*ReadNum> :exec readNumbers >>= println bad
                           <---- 由用户输入
*** ReadNum.idr:26:22:readNumbers 中 Main.case 块中的不匹配大小写
在 ReadNum.idr:26:22 ***
```

Idris 在检查readNumbers的整体时注意到了这一点：

```
*ReadNum> :total readNumbers_v2 Main.readNumbers 可
能不完整,原因是:ReadNum.idr:26:22 处 readNumbers 中的 Main.case 块,由于缺少案例,因
此不完整
```

记得检查整体! `readNumbers`的这个不完整的定义说明了为什么检查函数是总的很重要。虽然`readNumbers`类型检查成功,它仍然可能在运行时失败,因为失踪案件。

清单 5.6 展示了如何处理模式匹配中的其他可能性捆绑。除了绑定本身之外,您还可以在垂直之后提供替代匹配栏,显示如果Just上的默认匹配,函数的其余部分应如何进行`num1_ok`或`Just num2_ok`失败。该函数与清单 5.5 中的前一个函数具有完全相同的行为。

清单 5.6 从控制台读取和验证一对数字,简明扼要 (ReadNum.idr)

```
readNumbers : IO (也许 (Nat, Nat))
读取号码 =
只做 num1_ok <- readNumber |什么都没有 => 纯粹的没有
      只是 num2_ok <- readNumber |什么都没有 => 纯粹的没有
      纯 (只是 (num1_ok,num2_ok) )
```

读取第一个输入。如果它是无效,产生 Nothing 作为计算的结果。

读取第二个输入。如果它无效,则计算结果为 Nothing。

两个输入都是有效的,因此会产生一对读取的输入

这种形式的模式匹配绑定允许您在默认匹配中表达函数的预期有效行为(清单 5.6 中的仅`num1_ok`和`num2_ok`) ,处理替代匹配中的错误情况。

5.2.3 用循环编写交互式定义

现在您可以阅读、验证和响应用户输入,让我们将所有内容一起用循环编写交互式定义。

您可以通过编写递归定义来编写循环。例如,下一个清单,显示了一个倒计时函数,该函数计算一系列动作以显示倒计时,在显示每个数字之间有一秒钟的停顿。

清单 5.7 显示倒计时,每次迭代之间暂停一秒 (Loops.idr)

模块主要
System 模块包含几个与操作系统和环境交互的定义。
导入系统
你把它导入这里供我们睡觉。

倒计时:
倒计时: (秒:Nat) -> IO ()
倒计时 Z = putStrLn "起飞!"
倒计时 (S 秒)= 执行 putStrLn (显示 (S 秒))
睡眠 1000000
倒计时秒
usleep 描述了休眠给定微秒数的动作。

如果您尝试在REPL中使用 :exec 执行此操作,您将看到显示倒计时:

```
*Loops> :exec 倒计时 5 5
```

```
4
3
2
1
起飞!
```

您还可以检查此函数是否是总的,即保证在有限时间内为所有可能的输入产生结果:

```
*Loops> :total countdown Main.countdown
是 Total
```

通常,您可以编写一个交互式函数,通过对函数的最后一个操作进行递归调用来描述循环。在倒计时中,只要输入参数不是Z,程序在执行时将打印参数并等待一秒钟,然后再对下一个较小的数字进行递归调用。因此,循环的迭代次数由初始输入决定。因为这是有限的,倒计时必须最终终止,所以 Idris 报告它是全部的。

全面性和交互式程序 全面性检查是基于评估,而不是执行。因此,对IO程序进行全面检查的结果会告诉您 Idris 是否会产生有限的动作序列,但不会告诉您这些动作的运行时行为。

然而,有时迭代次数由用户输入决定。例如,您可以编写一个函数来继续执行倒计时,直到用户想要停止,如下所示。

清单 5.8 继续运行倒计时,直到用户不想再运行它 (Loops.idr)

```
countdowns : IO() countdowns =
do putStrLn "输入起始编号:"
  只需 startNum <- readNumber
  |什么都没有 => 做 putStrLn "无效输入"
    倒计时
  |如果用户输入 "y", 则
    进行递归调用
```

使用 readNumber 获得有效输入,如果输入无效则重新启动

这个函数不是全部的,因为不能保证用户会输入除y之外的任何东西,甚至提供任何有效的输入。

```
*Loops> :total countdowns Main.countdowns
由于递归路径可能不是总的: Main.countdowns
```

可能永远循环的交互式程序,例如倒计时(或更现实地说,服务器或操作系统)并不是全部的,至少如果我们将定义限制为终止程序的话。更准确地说,一个总函数必须要么终止,要么

保证在有限时间内产生一些无限输入的有限前缀。出色地
在第 11 章中进一步讨论。

练习



- 1**编写一个函数来实现一个简单的“猜数字”游戏。它应该有
以下类型:

猜测: (目标:Nat) -> IO ()

这里, target是要猜测的数字, guess的行为如下:

反复让用户猜一个数,并显示猜是否过分
高、过低或正确。

猜对后退出。

理想情况下,如果输入无效(例如,如果
它包含不是数字或超出范围的字符)。

- 2**实现一个在 1 到 100 之间选择一个随机数的main函数,然后
然后调用guess。

提示:作为随机数的来源,您可以使用time : IO Integer, defined
在系统模块中。

- 3**扩展guess以便计算用户已经猜出的猜数和dis
在读取输入之前播放该数字。

提示:将猜测的类型细化为以下:

猜测: (目标:Nat) -> (猜测:Nat) -> IO ()

- 4**实现您自己的repl和replWith 版本。请记住,您需要
使用不同的名称以避免与前奏曲中定义的名称发生冲突。

5.3 读取和验证依赖类型

在前面的章节中,您已经看到了几个具有依赖类型的函数,尤其是
使用Vect来表示其类型中向量的长度。这允许您在函数的类型中陈述关于输入形式的假设,并保证形式

其输出。例如,您已经看到zip,它将对应的元素配对
载体:

zip : Vect na -> Vect nb -> Vect n (a, b)

该类型表示以下内容:

假设 两个输入向量的长度相同, n。

保证 输出向量与输入向量的长度相同。

保证 输出向量将由第一个输入向量的元素类型和第二个输入向量的元素类型对组成。

Idris 检查每当调用函数时,参数是否满足假设,并且函数的定义是否满足保证。

到目前为止,我们一直在REPL测试诸如zip之类的功能。但实际上,函数的输入并非来自 Idris 类型检查器可用的这种精心控制的环境。在实践中,当一个完整的程序被编译和执行时,函数的输入将来自一些外部来源:可能是网页上的一个字段,或者是控制台上的用户输入。

当程序从某个外部源读取数据时,它不能对该数据的形式做出任何假设。相反,程序必须检查数据是否具有必要的形式。函数的类型准确地告诉您为了安全地评估它需要检查什么。

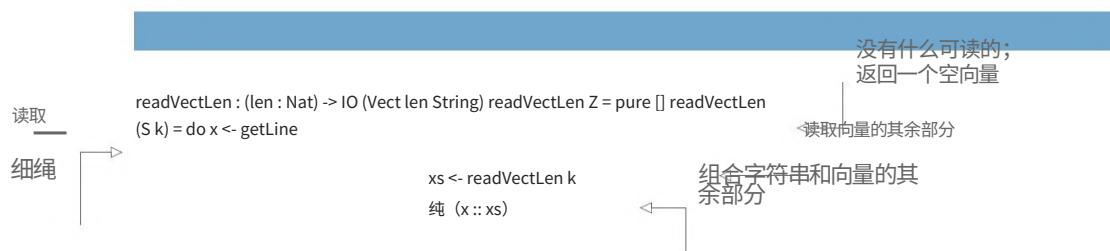
在本节中,我们将编写一个程序,从控制台读取两个向量,如果向量长度相同,则使用zip将对应的元素配对,然后显示结果。虽然这是一个简单的目标,但它展示了在交互式程序中使用依赖类型的几个重要方面,您将在本书后面看到更多这种形式的示例。您将简要了解程序的纯部分的类型如何告诉您需要在交互部分中检查什么,以及类型系统如何引导您进入需要进行错误检查的部分。

5.3.1 从控制台读取 Vect

作为第一步,您需要能够从控制台读取向量。因为向量在类型中表示它们的长度,所以您需要某种方式来描述您打算读取的向量的长度。一种方法是将长度作为输入,例如以下类型:

```
readVectLen : (len : Nat) -> IO (Vect len String)
```

此类型声明readVectLen将预期长度作为输入,并返回读取该长度字符串向量的操作序列。下面的清单显示了一种实现此功能的方法。



您可以通过以特定长度执行REPL中的readVectLen来尝试它,并且然后用printLn 打印结果:

```
*ReadVect> :exec readVectLen 4 >>= printLn
约翰
保罗
乔治
林戈
[ “约翰”、“保罗”、“乔治”、“林戈” ]
```

← 由用户输入 ← 伊德里斯的输出

这种方法的一个问题是您需要提前知道向量应该有多长,因为长度是作为输入给出的。相反,如果您想要

读取字符串直到用户输入一个空行?你无法提前知道有多少用户将输入的字符串,因此,您需要返回长度以及该长度的向量。

5.3.2 读取未知长度的 Vect

如果您从控制台读取一个以空行终止的向量,您将不知道如何许多元素将在结果向量中。在这种情况下,您可以定义一个新的不仅包含向量而且包含其长度的数据类型:

```
数据 VectUnknown : 类型 -> 类型在哪里
MkVect : (only : Nat) -> Vect only a -> VectUnknown a
```

在类型驱动开发中,我们的目标是表达我们对数据类型的了解;如果我们对数据一无所知,我们也需要以某种方式表达这一点。这是VectUnknown的目的;它包含向量及其长度,这意味着长度不必在类型中知道。

您可以在REPL 中构建一个示例:

```
*ReadVect> MkVect 4 [ “约翰”、“保罗”、“乔治”、“林戈” ]
MkVect 4 [ John , Paul , George , Ringo ]: VectUnknown String
```

事实上,你可以在表达式中留下下划线而不是给出长度,4,明确地,因为 Idris 可以从给定向量的长度推断出这一点:

```
*ReadVect> MkVect _ [ “约翰”、“保罗”、“乔治”、“林戈” ]
MkVect 4 [ John , Paul , George , Ringo ]: VectUnknown String
```

定义了 VectUnknown,而不是编写返回IO 的函数 (Vect len String),可以写一个返回IO (VectUnknown String) 的函数。这就对了不仅返回向量,还返回它的长度:

```
readVect : IO (VectUnknown 字符串)
```

这种类型表明readVect是一系列交互动作,它们产生一个未知长度的向量,这将在运行时确定。以下列表显示了一种可能的实现方式。

读取和验证依赖类型

清单 5.10 从控制台读取未知长度的Vect (ReadVect.idr)



要尝试此操作,您可以定义一个方便的函数printVect,它在控制台上显示VectUnknown的内容和长度:

```

printVect : 显示一个 => VectUnknown a -> IO ()
printVect (MkVect len xs)
  = putStrLn (显示 xs ++
              " (长度"
              ++ 显示长度 ++ " )")
  
```

然后,您可以尝试在REPL中读取一些输入:

```

*ReadVect> :exec readVect >>= printVect
约翰
保罗
乔治
林戈
  
```

由用户输入

用户输入的空行

伊德里斯的输出

在处理用户输入时,通常会有一些数据属性
直到运行时才能知道。向量的长度就是一个例子:一旦你读过
向量,你知道它的长度,从那里你可以检查它并推断它是如何产生的
与其他数据有关。但是对于任何依赖数据,您可能会遇到类似的问题
从用户输入中读取的类型,最好不要定义新类型(例如
VectUnknown)每次发生这种情况。相反,Idris 提供了一个更通用的解决方案,依赖对。

5.3.3 依赖对

你已经看过第 2 章中介绍的元组,它允许你组合不同类型的值,如下例所示:

```

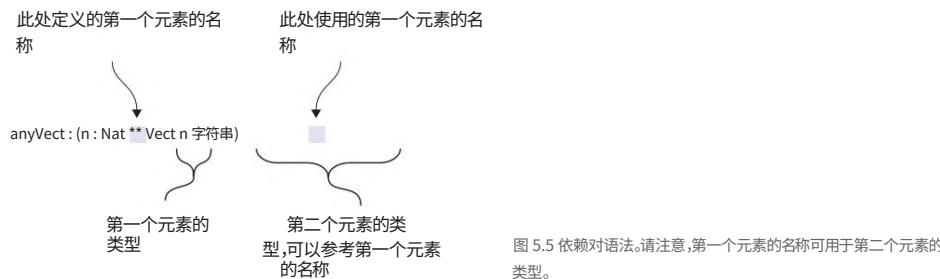
mypair : (整数,字符串)
mypair = (94, 页面)
  
```

依赖对是这种构造的一种更具表现力的形式,其中第二个的类型
可以从第一个元素的值计算出一对中的元素。例如:

```

anyVect : (n : Nat ** Vect n 字符串)
anyVect = (3 ** [ Rod , Jane , Freddy ])
  
```

依赖对是用**分隔的元素编写的。它们的类型使用与它们的值相同的语法编写,除了第一个元素有一个明确的名称(前面示例中的n)。图5.5说明了依赖对类型的语法。



如果Idris可以从第二个元素的类型推断出第一个元素的类型,您也可以经常省略它。例如:

```
anyVect : (n ** Vect n String) anyVect = (3 **  
[ Rod , Jane , Freddy ])
```

如果将值[Rod , Jane , Freddy]替换为一个孔,您可以看到第一个值3如何影响其类型:

```
anyVect : (n ** Vect n String) anyVect = (3 ** ?  
anyVect_rhs)
```

检查?anyVect_rhs的类型显示第二个元素必须是长度为3的向量,如第一个元素所指定:

```
anyVect_rhs : Vect 3 String
```

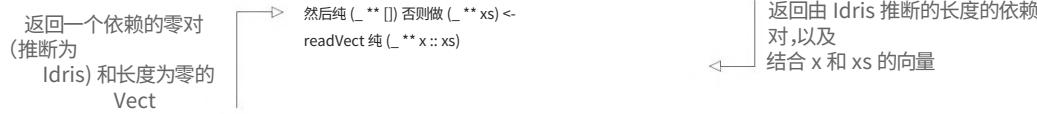
子表达式的**类型**请记住,当试图理解更大的程序列表时,您可以将子表达式替换为一个孔,例如**类型推断示例**中的?anyVect_rhs,以找出它们的预期类型以及范围内的任何局部变量的类型。

除了在5.3.2节中定义VectUnknown之外,您还可以定义一个函数,该函数通过返回长度的依赖对和该长度的向量来读取未知长度的向量。下面的清单显示了如何使用依赖对来定义readVect。

清单5.11 从控制台读取一个未知长度的Vect,返回一个依赖凹痕对(DepPairs.idr)

```
readVect : IO (len ** Vect len String) readVect = do x <- getLine if  
(x == )
```

返回构建一定长度的依赖对的动作,以及该长度的向量



同样,您可以在REPL上尝试一下。您可以使用println显示依赖对的内容:

```
*DepPairs> :exec readVect >= println
杆
简
弗雷迪
(3 ** [ "罗德"、"简"、"弗雷迪" ])
```

由用户输入
用户输入的空行
伊德里斯的输出

现在您已经能够从控制台读取任意长度和用户定义长度的向量,我们可以完成最初的目标,即编写一个程序来读取两个向量,并在它们的长度匹配时将它们压缩在一起。

5.3.4 验证 Vect 长度

如本节开头所述,我们的目标是编写一个执行以下操作的程序:

- 1 使用readVect读取两个输入向量。
- 2 如果它们的长度不同,则显示错误。
- 3 如果它们的长度相同,则显示将向量压缩在一起的结果。

该程序将在控制台上从用户那里获取输入,并在控制台上显示其结果。我们将其实现为zipInputs函数,如下所示:

- 1 类型 因为输入和输出完全在控制台,所以类型声明zipInputs不接受参数并返回交互操作:

压缩输入IO()

- 2 定义 定义函数的第一步是使用read Vect 读取两个输入。为定义的其余部分留一个洞, ?
zipInputs_rhs :

```
zipInputs : IO () zipInputs = do
  putStrLn "输入第一个向量 (空行到结尾) :
  (len1 ** vec1) <- readVect putStrLn "输入第二个
  向量 (空行到结尾) : (len2 ** vec2) <- readVect ?zipInputs_rhs
```

- 3 类型查看?zipInputs_rhs 的类型,您可以看到类型 (以及因此,已读取向量的长度) :

```
len2 : 晚上
vec2:Vect len2 字符串 len1:Nat
-----
vec1 : Vect len1 字符串
-----
zipInputs_rhs : IO ()
```

vec1的长度为len1, vec2的长度为len2;这些长度之间没有明确的关系。事实上,不应该有,因为它们是独立阅读的。但是,如果您查看zip 的类型,您会发现长度必须相同才能使用它:

```
zip : Vect na -> Vect nb -> Vect n (a, b)
```

4优化 在取得进展之前,您需要检查第二个向量的长度是否与第一个向量的长度相同。作为第一次尝试,您可以尝试以下操作:

```
如果 len1 == len2
    然后 ?ziplnputs_rhs1 否则 ?
        ziplnputs_rhs2
```

不幸的是,这无济于事。如果您查看?ziplnputs_rhs1 的类型,您会发现没有任何变化:

```
len1 : 晚上
len2 : 晚上
vec2 : Vect len2 字符串 vec1 : Vect len1 字符
串
-----
ziplnputs_rhs1 : IO ()
```

问题是 len1 == len2, Bool 的类型并没有告诉你操作本身的含义。就 Idris 而言, ==操作可以以任何方式实现 (您将在第 7 章中看到如何定义==)并且不一定保证len1和len2真的相等。

相反,您可以使用Data.Vect 中定义的以下函数:

精确长度: (len:Nat) -> (输入:Vect ma) -> Maybe (Vect len a)

这个函数接受一个长度len和一个任意长度的向量input 。如果输入向量的长度为len,则返回Just input ,其类型更新为Vect len a。否则,它返回Nothing。

实现EXACTLENGTH要实现精确长度,您需要一个比Bool更具表现力的类型来表示相等测试的结果。

您将在第 8 章看到如何做到这一点,我们将讨论Bool的一般限制。

使用exactLength,您可以按如下方式细化定义:

```
ziplnputs : IO () ziplnputs = do
    putStrLn "输入第一个向量 (空行到结尾) : "
    (len1 ** vec1) <- readVect putStrLn "输入第二个
    向量 (空行到结尾) : "
    (len2 ** vec2) <- readVect case exactLength len1 vec2 of
```

```
    什么都没有 => ?ziplnputs_rhs_1 只是 vec2  => ?
    ziplnputs_rhs_2
```

5 Refine - 对于zipInputs_rhs_1,输入的长度不同,因此您显示

一个错误:

```
case exactLength len1 vec2 of
    Nothing => putStrLn "向量长度不同"
    只是 vec2 => ?zipInputs_rhs_2
```

6 优化对于zipInputs_rhs_2,您有一个新向量vec2 ,它与vec2相同,但现在保证其长度与vec1的长度相同,您可以通过查看类型来确认:

```
len1 : 晚上
vec2 : Vect len1 字符串 len2:Nat

vec2 : Vect len2 字符串 vec1 : Vect len1 字符
串
-----
zipInputs_rhs_2 : IO ()
```

因此,您可以通过使用vec1和vec2 调用zip来完成定义,然后打印结果:

```
case exactLength len1 vec2 of
    Nothing => putStrLn "向量长度不同"
    只是 vec2 => println (zip vec1 vec2 )
```

作为参考,下面的清单给出了完整的定义。

清单 5.12 zipInputs (DepPairs.idr) 的完整定义

```
zipInputs : IO () zipInputs = do
    putStrLn "输入第一个向量 (空行到结尾) : "
    (len1 ** vec1) <- readVect putStrLn "输入第二个向量 (空行到结尾) : "
    (len2 ** vec2) <- readVect case exactLength len1 vec2 of
```

```
Nothing => putStrLn "向量长度不同"
只是 vec2 => println (zip vec1 vec2 )
```

练习

 在这些练习中,除了本章前面讨论的函数之外,您还会发现以下 Prelude 函数很有用: openFile、closeFile、fEOF、fGetLine 和 writeFile。使用:doc 找出它们各自的作用。此外,请参阅边栏“处理 I/O 错误”。

1 编写一个函数readToBlank : IO (List String),它从控制台,直到用户输入一个空行。

2 编写一个函数readAndSave : IO (),从控制台读取输入,直到用户输入一个空行,然后从控制台读取文件名并将输入写入该文件。

3编写一个函数readVectFile : (filename : String) -> IO (n ** Vect n String),它将文件的内容读入包含长度和该长度的Vect的依赖对。如果有任何错误,它应该返回一个空向量。

处理 I/O 错误练习中的许

多函数可能会使用Either 返回错误。首先,您可以假设使用模式匹配绑定的结果是成功的,如 5.2.2 节所述:

```
do Right h <- openFile 文件名 读取
    右行 <- fGetLine h {- 其余代码 -}
```

然后,为了使函数总计,使用同一部分中描述的符号处理错误:

```
do Right h <- openFile 文件名 读取
    | Left err => putStrLn (显示错误)
    右行 <- fGetLine h
    | Left err => putStrLn (show err) {- rest of code -}
```

5.4 总结

Idris 提供了一个通用的IO类型来描述交互动作。Idris 区分了纯函数的评估和交互动作的执行。REPL中的:exec命令执行交互式操作。

您可以使用do表示法对交互动作进行排序,它转换为
>>=运算符的应用。

您可以使用纯
功能。

Idris 提供了一个简明的符号,用于对 inter 的结果进行模式匹配
主动行动。

依赖类型表达了关于函数输入的假设,所以你需要
验证用户输入以检查它们是否满足这些假设。

依赖对允许您对两个值进行配对,其中第二个值的类型是根据第一个值计算的。你可以
使用依赖对来表示一个类型的参数,例如一个向量的长度,在用户输入一些输入之前是
未知的。

使用一流类型进 行编程

本章涵盖

使用类型级函数进行编程编写具有不同数量
参数的函数

使用类型级函数计算数据结构

完善更大的互动程序

在 Idris 中,正如您现在已经多次看到的那样,类型可以像任何其他语言构造一样被操作。例如,它们可以存储在变量中、传递给函数或由函数构造。此外,因为它们是真正的一流,表达式可以计算类型,并且类型也可以将任何表达式作为参数。您已经在实践中看到了这个概念的多种用途,特别是能够以数据的类型存储有关数据的附加信息,例如向量的长度。

在本章中,我们将探索更多利用类型的一流特性的方法。您将看到如何使用类型级函数为类型提供替代名称,以及如何根据其他数据计算函数的类型。在

特别是,您将看到如何编写类型安全的格式化输出函数printf。

对于printf,函数参数的类型(和偶数)是根据作为其第一个参数提供的格式字符串计算的。这种基于其他数据(在本例中为格式字符串)计算某些数据类型(在本例中为printf的参数)的技术通常很有用。这里有几个例子:

给定一个网页上的HTML表单,你可以计算出一个函数的类型

处理表单中的输入。

给定一个数据库模式,您可以计算该数据库上查询的类型。

作为这个概念的一个例子,你将看到如何使用类型级函数来优化我们在第4章末尾实现的数据存储。以前,你只能将数据存储为字符串,但为了获得更大的灵活性,你可能需要由用户描述的数据形式,而不是硬编码到程序中。使用类型级函数,您将使用描述数据形式的模式来扩展数据存储,并且您将使用该模式来计算用于解析和显示用户数据的函数的适当类型。通过这样做,您将了解更多关于在改进大型程序时使用孔来帮助纠正错误的信息。

首先,我们将了解如何在类型级别使用函数来计算类型。

6.1 类型级函数:计算类型

Idris最基本的特性之一是类型和表达式是同一种语言的一部分 两者使用相同的语法。你已经在第4章看到表达式可以出现在类型中。例如,在Vect上的append类型中,我们有n和m(都是Nats),得到的长度是n + m:

附加:Vect n elem -> Vect m elem -> Vect (n + m) elem

这里, n、m 和n + m都有Nat类型,也可以用在普通表达式中。

同样,您可以在表达式中使用类型,因此可以编写计算类型的函数。在两种常见情况下,您可能希望这样做:

为复合类型赋予更有意义的名称 例如,如果您有一个类型(Double, Double)将位置表示为2D坐标,您可能更愿意调用类型Position以使代码更具可读性和自文档化。允许函数的类型根据一些上下文信息而变化 例如,数据库查询返回的值的类型将根据数据库模式和查询本身而变化。

在第一种情况下,您可以定义类型同义词,为类型提供替代名称。在第二种情况下,您可以定义从某些输入计算类型的类型级函数(其中类型同义词是一种特殊情况)。在本节中,我们将看看两者。

6.1.1 类型同义词:为复杂类型赋予信息性名称

假设您正在编写一个处理复杂多边形的应用程序,例如绘图应用程序。您可以将多边形表示为每个角坐标的向量。例如,一个三角形可以按如下方式初始化:

```
tri : Vect 3 (Double, Double) tri = [(0.0, 0.0), (3.0, 0.0),
(0.0, 4.0)]
```

类型Vect 3 (Double, Double)准确地说明了数据的形式,这对机器很有用,但它没有向读者说明数据的目的是什么。相反,您可以使用表示为 2D 坐标的位置的类型同义词来优化类型:

位置:类型
位置 = (双倍,双倍)

在这里,您有一个名为Position的函数,它不接受任何参数并返回一个类型。这是一个普通的功能;它的声明或实施方式没有什么特别之处。现在,在任何可以使用类型的地方,都可以使用位置来计算该类型。例如,可以使用如下精炼类型定义三角形:

```
tri : Vect 3 位置
三 = [(0.0, 0.0), (3.0, 0.0), (0.0, 4.0)]
```

这种函数是类型同义词,因为它为其他类型提供了替代名称。

命名约定按照约定,我们通常对用于计算类型的函数使用首字母大写。

因为它们是普通函数,所以它们也可以带参数。例如,以下清单显示了如何使用类型同义词更清楚地表达Vect旨在表示多边形。

清单 6.1 使用类型同义词定义多边形 (TypeSynonym.idr)

```
导入数据.Vect
位置:类型
位置 = (双倍,双倍)
多边形:Nat -> 类型
多边形 n = Vect n 位置
tri : 多边形 3 tri = [(0.0, 0.0),
(3.0, 0.0), (0.0, 4.0)]
```

因为Polygon是一个普通函数,您可以在REPL 中对其进行评估:

```
*TypeSynonyms> 多边形 3
Vect 3 (双,双) : 类型
```

另外,请注意,如果您在REPL中评估tri , Idris将以评估形式显示tri 的类型。换句话说, REPL的求值同时求值表达式和类型:

```
*TypeSynonyms> tri [(0.0,  
0.0), (3.0, 0.0), (0.0, 4.0)] : Vect 3 (Double, Double)
```

另一方面,使用:t会显示tri 的类型:

```
*TypeSynonyms> :t tri  
多边形 3
```

最后,您可以看到当您尝试在 Atom 中使用表达式搜索以交互方式定义tri时会发生什么。将以下内容连同之前的Polygon 定义一起输入到 Atom 缓冲区中:

```
tri : 多边形 3 tri = ?tri_rhs
```

一般来说,交互式编辑工具,尤其是表达式搜索,都知道类型同义词是如何定义的,因此如果您尝试在tri_rhs 上进行表达式搜索,您将获得与直接将类型编写为Vect 3 (双倍,双倍) :

```
tri : 多边形 3 tri = [(?  
tri_rhs1, ?tri_rhs2), (?tri_rhs3, ?tri_rhs4), (?tri_rhs5, ?tri_rhs6)]
```

本节中定义的类型同义词Position和Polygon实际上只是碰巧用于计算类型的普通函数。这为您描述类型的方式提供了很大的灵活性,正如您将在本章的其余部分中看到的那样。

6.1.2 具有模式匹配的类型级函数

类型同义词是类型级函数的特例,这些函数可以在 Idris 期望类型的任何地方使用。就 Idris 而言,类型级函数并没有什么特别之处。它们是碰巧返回类型的普通函数,它们可以使用其他地方可用的所有语言结构。尽管如此,分开考虑它们是有用的,看看它们在实践中是如何工作的。

因为类型级函数是返回类型的普通函数,所以可以通过大小写来编写。例如,以下清单显示了一个从布尔输入计算类型的函数。您在第 1 章中看到了这个函数,尽管形式略有不同。

清单 6.2 一个从Bool计算类型的函数 (TypeFuns.idr)

```
StringOrInt : Bool -> 类型  
StringOrInt False = 字符串  
StringOrInt True = Int
```

使用它,您可以编写一个函数,其中返回类型是根据输入计算或取决于输入。使用StringOrInt,您可以编写返回任一类型的函数,具体取决于布尔标志。

作为一个小例子,您可以编写一个函数,该函数接受布尔输入,如果为False,则返回字符串“94”,如果为True,则返回整数94。从类型开始: 1类型首先给出类型声明,使用StringOrInt:

```
getStringOrInt : (isInt : Bool) -> StringOrInt isInt getStringOrInt isInt = ?getStringOrInt_rhs
```

如果您现在查看getStringOrInt_rhs的类型,您会看到:

```
isInt : 布尔值
-----
getStringOrInt_rhs : StringOrInt isInt
```

2定义 因为isInt出现在getStringOrInt_rhs 所需的类型中,所以对isInt进行大小写拆分会导致预期的返回类型根据每个大小写的isInt的具体值发生变化。 isInt上的大小写拆分导致:

```
getStringOrInt : (isInt : Bool) -> StringOrInt isInt getStringOrInt False = ?getStringOrInt_rhs_1
getStringOrInt True = ?getStringOrInt_rhs_2
```

3类型 - 查看新创建的孔的类型,您可以看到在每种情况下预期类型是如何变化的:

```
-----
getStringOrInt_rhs_1 : 字符串
-----
getStringOrInt_rhs_2 : 整数
```

在getStringOrInt_rhs_1 中,类型被细化为StringOrInt False ,因为 isInt 的模式为False ,其计算结果为String。然后,在getStringOrInt_rhs_2 中,类型被细化为StringOrInt True ,其计算结果为Int。

4细化要完成定义,您需要在每种情况下提供不同类型的值:

```
getStringOrInt : (isInt : Bool) -> StringOrInt isInt getStringOrInt False = 九十四  getStringOrInt
True = 94
```

[依赖模式匹配](#)(getStringOrInt示例说明了一种在使用依赖类型进行编程时通常很有用的技术:依赖模式匹配。这是指一个函数的一个参数的类型可以通过检查另一个参数的值(即通过大小写拆分)来确定的情况。在第4章定义zip时,您已经看到了一个例子,其中一个向量的形式限制了另一个向量的有效形式,你会看到更多。

类型级函数可以在任何需要类型的地方使用,这意味着它们可以
也可以用来代替参数类型。例如,您可以编写一个函数
将String或Int转换为规范的String表示,根据
一个布尔标志。该函数的行为如下:

如果输入是一个字符串,它返回带有前导和尾随空格的字符串
使用修剪删除。
如果输入是一个Int,它使用强制转换将输入转换为一个字符串。

[文档](#)请记住,您可以使用:t和:doc来检查
REPL中不熟悉的功能类型(例如修剪)。

您可以通过以下步骤定义此函数:

1类型 首先为valToString 编写类型,再次使用StringOrInt,但是
这次计算输入类型:

```
valToString : (isInt : Bool) -> StringOrInt isInt -> String
valToString isInt y = ?valToString_rhs
```

检查valToString_rhs 的类型,您将看到以下内容:

```
isInt : 布尔值
y : StringOrInt isInt
-----
valToString_rhs : 字符串
```

2定义 - 您可以通过isInt 上的大小写拆分来定义它。 y的类型是从isInt 计算出来的,所以如果你对
isInt 进行大小写拆分,你应该会看到精炼的类型
y在每种结果情况下:

```
valToString : (isInt : Bool) -> StringOrInt isInt -> String
valToString False y = ?valToString_rhs_1
valToString True y = ?valToString_rhs_2
```

3类型检查 valToString_rhs_1和valToString_rhs_2的类型
显示在每种情况下如何细化y的类型:

```
y : 字符串
-----
valToString_rhs_1 : 字符串
y : 整数
-----
valToString_rhs_2 : 字符串
```

4细化 要完成定义,请填写右侧以将y转换为 a
如果它是字符串,则修剪字符串,或者如果它是数字的字符串表示形式
是一个整数:

```
valToString : (isInt : Bool) -> StringOrInt isInt -> String
valToString False y = 修剪 y
valToString True y = cast y
```

本节中的简单示例getStringOrInt和valToString说明了一个可以在实践中更广泛地使用的技术。你会看到一些更实用的本章后面的示例:使用格式字符串计算格式化的类型输出,以及一个更大的示例,允许用户计算模式,扩展数据您在第4章中看到的商店。

但是,您可以使用类型级表达式实现更多目标。事实类型是一流的意味着不仅可以像其他类型一样计算类型值,而且任何表达式形式都可以出现在类型中。

6.1.3 在类型中使用 case 表达式

可以在函数中使用的任何表达式也可以在类型级别使用,并且反之亦然。例如,您可以在类型中留下漏洞,而您对函数的要求发展,或者你可以使用更复杂的表达形式比如案例。让我们通过使用case表达式来简要地看一下它是如何工作的在valToString类型而不是单独的StringOrInt函数中:

1 种类型 首先给出valToString 的类型,但因为您不立即早知道输入类型,就可以留个坑:

```
valToString : (isInt : Bool) -> ?argType -> String
```

2 定义 即使你的类型不完整,你仍然可以继续在isInt 上按大小写定义,因为你至少知道isInt是布尔:

```
valToString : (isInt : Bool) -> ?argType -> String
valToString False y = ?valToString_rhs_1
valToString True y = ?valToString_rhs_2
```

3 类型检查valToString_rhs_1和valToString_rhs_2的类型,但是,表明您还不知道y的类型:

```
    和: ?argType
-----
valToString_rhs_1 : 字符串
```

4 Refine 在这个阶段,您需要使用?argType 的更多细节来优化类型。您可以使用case表达式填写?argType :

```
valToString : (isInt : Bool) -> (case
  _           的
  case_val => ?argType) -> 字符串
valToString False y = ?valToString_rhs_1
valToString True y = ?valToString_rhs_2
```

5 Refine 记住,当Idris生成一个case表达式时,它会留下一个下划线代替case表达式将分支的值。你会需要在程序编译之前填写。您可以从输入isInt 计算参数类型:

```
valToString : (isInt : Bool) -> (case isInt of
  case_val => ?argType) -> 字符串
```

```
valToString False y = ?valToString_rhs_1
valToString True y = ?valToString_rhs_2
```

6 Refine case_val上的大小写拆分为您提供两个可能的值isInt可以拿：

```
valToString : (isInt : Bool) -> (case isInt of
                                         假 => ?argType_1
                                         真 => ?argType_2) -> 字符串
valToString False y = ?valToString_rhs_1
valToString True y = ?valToString_rhs_2
```

然后，您可以以与您的实现相同的方式完成类型

StringOrInt,用String精炼?argType_1和用Int精炼?argType_2：

```
valToString : (isInt : Bool) -> (case isInt of
                                         假 => 字符串
                                         True => Int) -> 字符串
valToString False y = ?valToString_rhs_1
valToString True y = ?valToString_rhs_2
```

7 类型检查?valToString_rhs_1和?valToString_rhs2现在将显示

你输入y的新类型，从第一个参数计算：

```
y : 字符串
-----
valToString_rhs_1 : 字符串
y : 整数
-----
valToString_rhs_2 : 字符串
```

8 Refine 最后，既然您知道每种情况下输入y的类型，您就可以像以前一样完成定义：

```
valToString : (isInt : Bool) -> (case isInt of
                                         假 => 字符串
                                         True => Int) -> 字符串
valToString False y = 修剪 y
valToString True y = cast y
```

总体和类型级函数

一般来说，最好以与普通函数完全相同的方式来考虑类型级函数，就像我们目前所做的那样。不过，情况并非总是如此。有一个

了解一些有用的技术差异：

类型级函数仅在编译时存在。没有Type的运行时表示，也没有办法直接检查Type，例如模式匹配。只有总计的函数才会在类型级别进行评估。一个函数

is not total 可能不会终止，或者可能不会涵盖所有可能的输入。所以，为了确保类型检查本身终止，不完全的函数是在类型级别被视为常量，并且不进一步评估。

6.2 定义具有可变数量参数的函数

您可以使用类型级函数根据其他一些已知输入计算类型。

鉴于函数类型本身就是类型,这意味着您可以编写函数

根据其他一些输入,具有不同数量的参数。这是相似的

到一些支持可变长度参数列表的其他语言,但在类型中具有额外的精度,因为您使用一个参数的值来计算

其他类型。

在本节中,我们将看到几个例子来说明它是如何工作的: 作为一个介绍性的例子,

我们将编写一个函数来添加一个数字序列,其中第一个参数是一个用于计算类型的数字一个需要这么多输入的函数。

我们将使用相同的技术来编写printf函数的变体,该函数在其第一个参数中使用格式说明符生成格式化的输出字符串
描述后面论证的形式。

6.2.1 一个加法函数

首先,我们将定义一个加法函数

将一系列数字直接给出为

函数参数。它的行为特点是

三个例子如图 6.1 所示。 adder numargs val形式的表达式计算一个

接受numargs附加参数的函数,
它们被添加到初始值val。

与类型驱动开发中的往常一样,您将通过写它的类型开始写加法器,但是在这个

如果类型不是您可以构造的东西

直接地;加法器的类型取决于

第一个参数的值。对于示例

如图 6.1 所示,您正在寻找以下内容

类型:

加法器 0 : Int -> Int

加法器 1 : Int -> Int -> Int

加法器 2 : Int -> Int -> Int -> Int

...

因为类型是一等的,并且这种类型根据某些值而有所不同,所以您将

能够使用类型级函数计算它。你可以写一个AdderType函数

具有所需的行为。 AdderType的名称遵循类型级函数的名称以首字母大写字母命名的约定,它表示它是

用于计算加法器的类型。

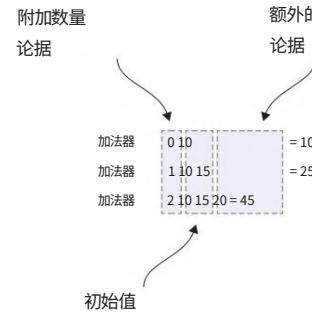


图 6.1 具有可变数量参数的加法函数的行为

您可以使用Nat来给出参数列表的长度,因为您可以方便地对其进行大小写拆分,并且因为具有负长度的参数列表将毫无意义。下面的清单给出了AdderType 的定义。

清单 6.3 计算加法器 n (Adder.idr)类型的函数

```
AdderType : (numargs : Nat) -> 类型
AdderType Z = Int
AdderType (S k) = (next : Int) -> AdderType k
```

返回一个函数,它接受一个整数并构造加法器类型的
其余部分,它接受 k 个进一步的参数

没有更多的
函数参数

现在可以通过将其第一个参数传递给AdderType来计算加法器的类型。

第二个参数是初始值,我们将其称为acc作为“累加器”的缩写:

```
adder : (numargs : Nat) -> (acc : Int) -> AdderType numargs
```

因为你在 AdderType 中对 numargs 进行大小写拆分计算类型,所以可以通过对numargs进行大小写拆分,将adder的定义写成对应的结构,这样AdderType就会针对每个 case 进行细化。您可以通过以下步骤实现它:

1 定义添加一个骨架定义,然后在第一个参数上进行大小写:

```
adder : (numargs : Nat) -> (acc : Int) -> AdderType numargs adder Z acc = ?adder_rhs_1 adder (S k) acc = ?adder_rhs_2
```

2 Refine - 如果附加参数numargs的数量为Z,则返回
直接累加器:

```
adder : (numargs : Nat) -> (acc : Int) -> AdderType numargs adder Z acc = acc
adder (S k) acc = ?adder_rhs_2
```

3 类型对于adder_rhs_2,检查孔的类型表明您需要提供一个函数:

```
k : 晚上
会计:国际
-----
adder_rhs_2 : Int -> AdderType k
```

这是一个函数类型,因为在AdderType中,当numargs匹配非零Nat 时,预期的类型是函数类型。

4 Refine 产生AdderType k类型的唯一方法通常是调用adder并以k作为第一个参数,因此这种类型暗示您需要对adder 进行递归调用。这是完整的定义:

定义具有可变数量参数的函数

```
adder : (numargs : Nat) -> (acc : Int) -> AdderType numargs adder Z acc = acc adder (S k) acc = \next
=> adder k (next + acc)
```

既然您已经有了一个完整且有效的定义,那么考虑如何改进类型或定义本身是一个好主意。例如,加法器可以在它添加的数字类型中通用。

清单 6.4 显示了加法器函数的一个稍微改进的版本,它适用于任何数字类型,而不仅仅是Int。它通过将额外的Type参数传递给AdderType,然后将其限制为加法器类型中的数字类型来实现这一点。

清单 6.4 适用于任何数字类型的通用加法器 (Adder.idr)

```
→ AdderType : (numargs : Nat) -> 类型 -> 类型
AdderType Z numType = numType
AdderType (S k) numType = (next : numType) -> AdderType k numType

adder : Num numType => (numargs :
    Nat) -> numType -> AdderType numargs numType adder Z acc = acc adder (S k) acc = \next
=> adder k (next + acc)
```

这里的类型是您将要添加在一起的参数的类型。

限制可以添加到数字类型的类型,使用Num numType 然后将 numType 传递给 AdderType

adder函数说明了定义具有可变数量参数的函数的基本模式:您已经编写了一个AdderType函数来计算所需的加法器类型,给定一个加法器的输入。该模式也可以应用于更大的定义,正如您现在将在我们定义用于格式化输出的函数时看到的那样。

6.2.2 格式化输出:类型安全的 printf 函数

具有可变数量参数的函数的最大示例是printf,在 C 和其他一些语言中可以找到。它在给定格式字符串和可变数量的参数(由格式字符串确定的情况下生成格式化输出。格式字符串本质上给出了一个要输出的模板字符串,由剩余的参数填充。本质上, printf的整体结构与adder 相同,使用格式字符串计算后面参数的类型。

图 6.2 显示了一些描述printf行为的示例。请注意,我们的printf版本不是向控制台生成输出,而是返回一个字符串。

描述附加参数的格式字符串
串

附加论点

打印	“你好! ”	= “你好! ”
打印	“答案:%d”42	= “答案:42”
打印	%s number %d	Page 94 = Page number 94

图 6.2 具有不同格式字符串的printf函数的行为

在这些格式字符串中,指令%d代表int; %s代表字符串;和其他任何内容都按字面意思打印。

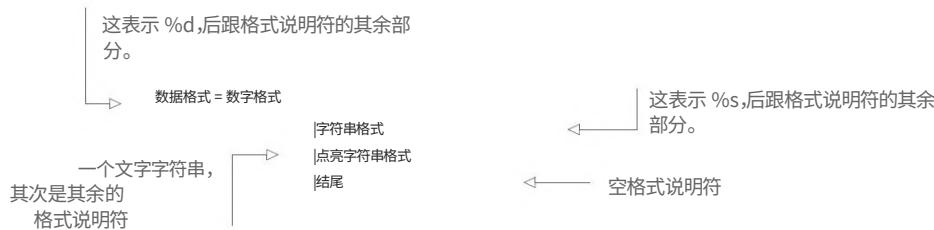
按照我们通常的类型、定义、改进过程,我们将首先考虑一个printf的类型。与加法器一样,我们将从特征示例的类型开始然后研究如何编写一个计算这些类型的函数。这些是图 6.2 中示例的类型:

```
printf 你好! : 细绳
printf Answer: %d : Int -> String
printf %s number %d : 字符串 -> 整数 -> 字符串
```

格式化字符串在C 提供的printf的完整实现中,可用的指令比%d和%s多得多。此外,该可以以各种方式修改指令以进一步指示输出如何应该格式化(例如数字中的前导零)。这样的细节但是,不要在类型级函数的讨论中添加任何内容,所以我们将在这里省略它们。

要获取printf 的类型,您需要使用格式字符串来构建预期的参数的类型。不用直接处理字符串,可以写一个数据类型描述可能的格式,如下所示。

清单 6.5 将格式字符串表示为数据类型 (Printf.idr)



这在格式字符串的解析和处理之间提供了一个清晰的分离,就像我们在第 4 章中将命令解析到数据存储时所做的那样。

例如, Str (点亮 “ = (Number End))将表示格式字符串 %s = %d ,如图 6.3 所示。



图 6.3 将格式字符串转换为格式描述

定义具有可变数量参数的函数

中间类型在类型驱动的

开发过程中,我们经常从数据类型之间的转换来考虑函数。因此,我们经常定义中间类型,例如本节中的 Format 来描述计算的中间阶段。

这样做的主要原因有两个:

仅从类型来看, String 的含义并不明显。函数类型 Format -> Type 比函数类型 String -> Type 具有更精确的含义,因为很明显输入必须是格式规范而不是任何 String。 定义一个中间数据类型让我们可以访问更多的交互式编辑

荷兰国际集团的特点,特别是案例拆分。

目前,我们将直接使用 Format;稍后我们将定义从 String 到 Format 的转换。以下清单显示了如何根据格式说明符计算 printf 的类型。

清单 6.6 从格式说明符计算 printf 的类型 (扩展 Printf.idr)

Number 指令意味着您的 printf 函数将需要另一个 Int 参数。

PrintfType : 格式 -> 类型
 ↳ PrintfType (Number fmt) = (i : Int) -> PrintfType fmt
 ↳ PrintfType (Str fmt) = (str : String) -> PrintfType fmt
 ↳ PrintfType (Lit str fmt) = PrintfType fmt
 ↳ PrintfType 结束 = 字符串

Str 指令意味着您的 printf 函数将需要另一个 String 参数。

这给出了 printf 的返回类型。

这里不需要额外的参数,因为你有一个文字字符串。您可以从格式的其余部分计算类型。

清单 6.7 定义了一个帮助函数,用于从 Format 构建一个 String,以及由 PrintfType 计算的任何必要的附加参数。这就像加法器一样,使用累加器来构建结果。

清单 6.7 printf 的辅助函数,从格式说明符 (Printf.idr) 构建字符串

String 是一个累加器,您可以在其中构建要返回的 String。

printfFmt : (fmt : Format) -> (acc : String) -> PrintfType fmt
 ↳ printfFmt (Number fmt) acc = \i => printfFmt
 ↳ printfFmt (Str fmt) acc = \str => printfFmt
 ↳ printfFmt (Lit lit fmt) acc = printfFmt
 ↳ printfFmt (acc ++ lit) printfFmt End acc = acc

PrintfType 从 Number fmt 计算出一个函数类型,所以这里需要构建一个函数。

最后,没有更多的参数要读取,也没有更多的文字输入,所以返回累加器。

记住交互式编辑!例如清单 6.6 中的示例

和 6.7,不要直接输入它们。相反,使用交互式编辑 Atom 中的工具来尝试自己重建它们。这样做时,请确保你看看任何孔的类型,看看表达式搜索有多远可以带你。

最后,实现一个将String作为格式说明符而不是格式结构,您需要能够将字符串转换为格式。以下清单定义了执行此转换的顶级printf函数。

清单 6.8 printf的顶级定义,从String到Format的转换 (Printf.idr)

在此处使用 List Char 而不是 String,以便您可以轻松匹配单个字符。

strCons 从初始值构建一个字符串字符和字符串的其余部分。

```
→ toFormat : (xs : List Char) -> 格式
toFormat [] = 结束
toFormat (% :: d :: chars) = Number (toFormat chars)
toFormat (% :: s :: chars) = Str (toFormat chars)
toFormat (% :: chars) = 点亮 % (toFormat chars)
toFormat (c :: chars) = case toFormat chars of
    点亮 点亮字符 => 点亮 (strCons c 点亮) 字符
    fmt => 点亮 (strCons c ) fmt
```

```
printf : (fmt : String) -> PrintType (toFormat (unpack fmt))
printf fmt = printfFmt
```

← 使用下划线 (_) 作为格式,因为 Idris 可以从类型推断出它必须是 toFormat (解压 fmt) 。

练习

1一个nxm矩阵可以用Double 的嵌套向量表示。定义类型同义词： Matrix : Nat -> Nat -> Type。

您应该能够使用它来定义以下矩阵：

```
测试矩阵:矩阵 2 3
测试矩阵 = [[0, 0, 0], [0, 0, 0]]
```

2扩展printf以支持Char和Double 的格式化指令。

您可以在REPL上测试您的答案,如下所示：

```
*ex_6_2> :t printf %c %f
printf %c %f :字符 -> 双精度 -> 字符串

*ex_6_2> printf %c %f      X 24
" X 24.0" :字符串
```

3您可以将向量实现为嵌套对,嵌套计算从

长度,如本例所示：

```
TupleVect 0 ty = ()
TupleVect 1 ty = (ty, ())
TupleVect 2 ty = (ty, (ty, ()))
...
...
```

使用模式增强交互式数据存储

定义实现此行为的类型级函数 TupleVect。记住从 TupleVect 的类型开始。

当您有正确答案时,以下定义将是有效的:

```
测试:TupleVect 4 Nat
测试 = (1,2,3,4,())
```

6.3 使用模式增强交互式数据存储

在我们在第 4 章开发的交互式数据存储中,您可以添加字符串到内存存储并通过索引检索它们。但是,如果您想存储更复杂的数据怎么办?如果您希望数据的形式由

用户在输入任何数据之前,而不是在程序本身中硬编码?

在本章的其余部分,我们将通过添加模式来扩展数据存储来描述数据的形式。我们将通过用户输入确定模式,我们将使用类型级别函数来计算数据的正确类型。图 6.4 显示了两个不同的数据存储,具有不同的模式,我们将能够用我们的扩展系统表示。
架构 1 在顶部显示了一个存储,该存储要求数据为(Int, String) 类型,下面的模式 2 显示了一个要求数据类型为(String, String, Int)。相反,在第 4 章开发的数据存储中,模式是有效地总是字符串。

模式 1: (整数,字符串)	
0	(11, “阿姆斯特朗、奥尔德林和柯林斯”)
1	(17, “塞尔南、埃文斯和施密特”)
2	(8,《博尔曼、洛弗尔和安德斯》)

模式 2: (字符串,字符串,整数)	
0	(“雨狗”, “汤姆等待”,1985)
1	(“泰恩河上的雾”, “林迪斯法恩”,1971)
2	(“洪水”, “他们可能是巨人”,1990)

图 6.4 两种不同的数据存储,具有不同的模式。
模式 1 要求数据的类型为(Int, String),模式 2 要求数据的类型为(String, String, Int)

与扩展系统的典型交互可能如下进行。注意我们在输入任何数据之前描述了模式。

```
命令 schema String String Int
好的
指挥部:添加 “Rain Dogs” “Tom Waits”1985
编号 0
命令 添加 “泰恩河上的雾” “林迪斯法恩”1971
编号 1
命令 :得到 1
“泰恩河上的雾”, “林迪斯法恩”,1971
命令 :退出
```

我们将从第 4 章中实现的现有数据存储开始，并根据需要改进整个系统，而不是从头开始。我们将采用这种方法：

- 1 优化 DataStore 的表示以包含模式描述。这种改进将不可避免地意味着我们的程序不再进行类型检查。
- 2 纠正所有错误，为更复杂的错误引入漏洞。
- 3 填补漏洞以完成实现，并添加任何进一步的功能由于改进了类型，现在支持。

6.3.1 细化 DataStore 类型

目前，DataStore 仅支持存储字符串。我们在第 4 章中使用以下类型实现了它：

```
数据数据存储:键入位置
MkData : (size : Nat) -> (items : Vect size String) -> DataStore
```

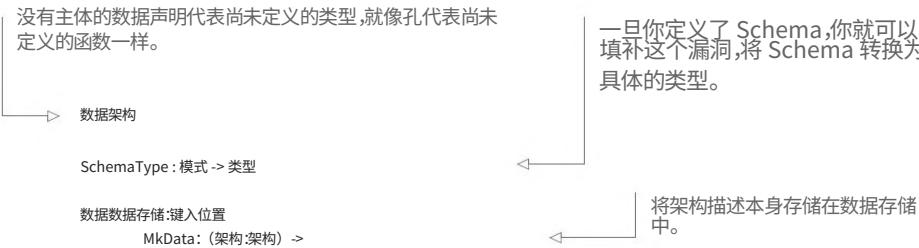
我们不想为商店中的商品使用 Vect 大小字符串，而是希望这些商品的类型灵活。一种自然的方法可能是将其细化为 DataStore 的通用版本，通过提供存储中数据类型的模式对其进行参数化：

```
数据数据存储:类型-> 键入位置
MkData : (size : Nat) -> (items : Vect size schema) -> DataStore schema
```

注释代码部分 当您在处理改进的 DataStore 时，您将不可避免地破坏程序的其余部分，这些部分将不再进行类型检查。因此，我建议您注释掉其余代码（将其放在{-和-}之间），直到您完成新的 DataStore 类型并准备好继续。

我们希望用户能够描述甚至可能更新模式，但是如果我们对模式类型进行参数化，模式将在类型中固定。相反，我们将创建一个用于描述模式的数据类型，以及一个用于将模式描述（可能由用户在运行时给出）转换为具体类型的类型级函数。下面的清单显示了精炼的 DataStore 类型的概要。

清单 6.9 细化的 DataStore 类型的大纲，带有 Schema 描述和 Schema 到未定义的具体类型的转换 (DataStore.idr)



(大小:Nat) -> (项目Vect 大
小 (SchemaType 模式)) ->
数据存储

根据模式计算商店中所
需的项目类型。

根据用户给出的定义,我们将模式定义为字符串和整数的某种组合。用户将提供Schema,我们将使用类型级函数SchemaType将Schema转换为某种具体类型。

下一个清单显示了如何使用类型级函数SchemaType来描述模式并将它们转换为 Idris类型。

清单 6.10 定义Schema,并将Schema转换为具体类型



您可以尝试使用此方法为前面图 6.4 中所示的两个示例商店定义模式：

```
*DataStore> SchemaType (SInt .+. SString)
(整数,字符串) :类型
```

```
*DataStore> SchemaType (SString .+. SString .+. SInt)
(字符串,字符串,整数) :类型
```

声明运算符您可以通过
赋予它们的固定性和优先级来定义新的运算符。在清单 6.10 中,你有这个：

中缀 5 .+。

这引入了一个优先级为 5 的新右关联中缀运算符 (infixr)。通常,使用关键字 infixl (用于左关联运算符)、
infixr (用于右关联运算符)或中缀 (用于非关联运算符)引入运算符运算符),后跟优先级和运算符列表。

甚至算术和比较运算符也是这样定义的,而不是内置语法。它们在前奏曲中介绍如下:

```
中缀 5 ==, /= 中缀 6 <, <=, >, >=
中缀 7 <<, >>
```

(续) 中缀 8 +, -
中缀 9 *, /

列表上的 :: 和 ++ 运算符也在 Prelude 中定义, 声明如下:

中缀 7 ::, ++

新的 DataStore 类型不仅允许您存储存储的大小和内容, 还可以将存储内容的结构描述存储为模式。

以前, 每个条目总是一个字符串, 但现在表单由用户确定。

现在, 因为您更改了 DataStore 的定义, 您还需要
更改访问它的功能。

6.3.2 使用数据存储的记录

为了能够使用更新后的 DataStore 类型, 您需要重新定义函数大小和项目以将相关字段投影到结构之外。

size 的定义和前面的定义类似:

```
大小 : DataStore -> Nat 大小 (MkData 模
式 大小 项目 ) = 大小
```

但是, 要定义项目, 您还需要编写一个函数来将模式投射到商店之外, 因为您需要知道模式描述才能知道商店中项目的类型。

```
架构 : 数据存储 -> 架构
架构 (MkData 架构 大小 项目 ) = 架构

items : (store : DataStore) -> Vect (size store) (SchemaType (schema store)) items (MkData schema size items ) = items
```

编写这样的投影函数, 本质上是从记录中提取字段, 很快就会变得乏味。取而代之的是, Idris 提供了一种用于定义记录的符号, 这会导致自动生成用于从记录中投影字段的函数。

您可以按如下方式定义 DataStore。

清单 6.11 将 DataStore 实现为记录, 并自动生成 投影函数

记录 DataStore 的位置

```
构造函数 MkData
时间表 : 时间表
尺寸 : 自然
项目 : Vect 大小 (SchemaType 模式)
```

← 命名 DataStore 的数据构造函数

声明字段, 自动生成同名投影函数

记录声明引入了一种新的数据类型,很像数据声明,但有两个不同之处:

只能有一个构造函数。字段产生投影函数,由字段类型自动生成。

对于DataStore,您可以使用:doc查看MkData数据构造函数的类型和从字段生成的投影函数:

```
*数据存储> :doc 数据存储
记录数据存储

构造函数:
MkData: (架构 架构) ->
    (大小:Nat) -> (项目,Vect 大
    小 (SchemaType 模式) ) -> 数据存储

预测:
架构 : (rec : DataStore) -> 架构

大小: (rec:数据存储) -> Nat

项目: (rec:DataStore) ->
    Vect (size rec) (SchemaType (schema rec))
```

您可以通过在REPL中创建一个简单的测试记录来尝试此操作:

```
*DataStore> :let teststore = (MkData (SString .+ SInt) 1 [( Answer , 42)])
*数据存储> :t 测试存储
测试存储:数据存储
```

接下来,您可以从该测试记录中投影模式、大小和项目列表:

```
*DataStore> 模式测试存储
字符串 .+。 SInt:架构

*DataStore> 大小测试存储
1:纳特

*DataStore> 项目测试存储
[( Answer , 42)]:Vect 1 (String,Int)
```

记录实际上比在这个小例子中看到的要灵活得多。除了投影字段的值外,Idris 还提供了一种用于设置字段和更新记录的语法。当我们在第 12 章讨论使用状态时,你会学到更多关于记录的知识。

6.3.3 使用孔纠正编译错误

现在您有了DataStore的新定义,使用它的旧程序将不再进行类型检查,因为它依赖于旧定义。那么,细化数据存储程序的下一步是更新定义,以便整个程序

再次进行类型检查。这并不一定意味着完成该计划；在这个阶段，可以通过插入稍后填充的孔来解决类型错误。

早些时候，我建议在定义 DataStore 之后暂时注释掉代码，这样您就可以处理细化的定义而不必担心编译错误。现在，我们将完成程序的其余部分，在 Idris 给我们的类型错误的指导下，一点点地取消注释定义并修复它们。

首先，让我们取消注释 addToStore，之前定义如下：

```
addToStore : DataStore -> String -> DataStore addToStore (MkData size store)
newitem = MkData
          _____ (addToData 存储)
在哪里
addToData : Vect oldsize String -> Vect (S oldsize) String addToData [] = [newitem] addToData (item :: items)
= item :: addToData items
```

在重新加载时，通过在 Atom 中使用 Ctrl-Alt-R 或在 REPL 中使用:r 命令，Idris 报告如下：

```
数据存储.idr:21:1-11:
检查 addToStore 的左侧时：检查 Main.addToStore 的应用程序时：Vect 大小
(SchemaType 模式) -> DataStore (MkData 模式大小的类型)之间的类型不匹配
```

和

数据存储（预期类型）

这里的第一个问题是您向 MkData 添加了一个参数；它现在需要一个模式以及一个大小和一个项目向量。您可以通过向 MkData 添加模式参数来纠正此问题：

```
addToStore : DataStore -> String -> DataStore addToStore (MkData schema size
store) newitem
          = MkData 模式 (addToData 存储)
在哪里
addToData : Vect oldsize String -> Vect (S oldsize) String addToData [] = [newitem] addToData (item :: items)
= item :: addToData items
```

伊德里斯现在报道

```
类型不匹配
Vect 大小 (SchemaType 模式) (存储类型)
和
Vect 大小字符串 (预期类型)
```

问题是数据存储不再仅存储字符串，而是存储模式描述的类型。您可以通过更改 addToStore 和 addToData 的类型来纠正此问题，以便它们使用正确的类型。 addToStore 的类型正确定义显示在以下清单中。

使用模式增强交互式数据存储

清单 6.12 使用改进的DataStore类型的addToStore的更正定义

```

addToStore : (store : DataStore) -> SchemaType (schema store) -> DataStore addToStore (MkData schema size store) newitem
    = MkData 模式 (addToData 存储)
在哪里
addToData : Vect oldsize (SchemaType schema) ->
    Vect (S oldsize) (SchemaType schema) addToData [] = [newitem]
    addToData (item :: items) = item :: addToData items

```

根据商店中定义的模式计算您使用 SchemaType 添加的项目的类型

这里的名称模式是指在前面的子句中绑定的名称

如果您继续一个一个地取消注释定义,您将遇到的下一个错误是在getEntry 中。它目前定义如下,错误的行被标记。

清单 6.13 旧版getEntry,应用index报错

```

getEntry : (pos : Integer) -> (store : DataStore) ->
    也许 (字符串,数据存储)
getEntry 邮政商店
= let store_items = items store in case integerToFin pos (size
    store) of
    什么都没有 => 只是 ( “超出范围\n” ,存储)
    Just id => Just (index id (items store)
        ++ \n ,存储)

```

应用索引时出现错误,因为存储不再将项目表示为包含字符串的Vect。

问题出在最后一行,您从商店中提取商品,因为您将商店视为包含字符串的Vect。以下是伊德里斯报告的内容:

检查函数 Data.Vect.index 的应用程序时,Vect 之间的类型不匹配 (大小存储)

(SchemaType (模式存储)) (项目存储类型)
和
Vect (大小存储)字符串 (预期类型)

问题出在索引的应用中,它不再返回一个字符串。您可以通过插入一个孔将index的结果转换为String来临时纠正此错误,如下所示。

清单 6.14 通过插入一个洞来纠正getEntry以将商店的内容转换为可显示的字符串

```

getEntry : (pos : Integer) -> (store : DataStore) ->
    也许 (字符串,数据存储)
getEntry 邮政商店
= let store_items = items store in case integerToFin pos (size
    store) of
    什么都没有 => 只是 ( “超出范围\n” ,存储)
    Just id => Just (?display (index id (items store)) ++ \n ,
        店铺)

```

?display 洞,填完后会是一个函数,将 index的结果,即SchemaType (schema store),转换成可以显示的String。

如果你检查显示的类型,你会看到你需要填补这个洞的函数的类型,将一个SchemaType (模式存储)转换为一个字符串:

```
存储:数据存储
id : Fin (size store) pos : Integer
store_items : Vect (size store) (SchemaType
(schema store))
-----
显示:SchemaType (模式存储) → 字符串
```

我们很快就会回到?display。目前, getEntry再次进行类型检查。下一个错误在processInput中。这是当前的定义。

清单 6.15 旧版processInput,在应用中出现错误

[添加到存储](#)

```
processInput : DataStore -> String -> Maybe (String, DataStore) processInput store input = case parse input of
```

```
Nothing => Just ( Invalid command\n , store)
只是 (添加项目)=>
    只是 ( “身份证” ++ show (size store) ++ \n , addToStore store item)
Just (Get pos) => getEntry pos store
只是退出 => 没有
```

这里的 addToStore 应用程序有一个错误,因为它向它传递了一个字符串,它现在需要一个由模式类型描述的条目。

这个定义有一个类似getEntry的错误,说明你有一个String但是Idris 期望一个SchemaType (模式存储) :

检查函数 Main.addToStore 的应用程序时:类型不匹配

```
字符串 (项目类型)
和
SchemaType (模式存储) (预期类型)
```

一种可能的解决方法是,与getEntry一样,在processInput中添加一个孔,用于将String转换为适当的SchemaType (模式存储) :

只是 (“身份证” ++ show (size store) ++ \n , addToStore store (?convert item))

或者,您可以细化Command的定义,使其仅表示有效命令,这意味着任何无效的用户输入都会导致解析错误。

我们将采用这种方法,因为它涉及定义更精确的中间类型,因此您将尽早检查输入的有效性。

为此,请通过模式描述参数化Command ,并更改Add命令,使其直接采用SchemaType而不是String输入。

这是命令的精确定义。

使用模式增强交互式数据存储

清单 6.16 命令,细化为由数据存储中的模式参数化

→ 数据命令:架构-> 输入位置

```
添加:SchemaType 模式 -> 命令模式
获取:整数 -> 命令模式
退出:命令模式
```

命令由模式描述参数化,它给出了可以添加到存储
中的条目的形式。

Add 的类型现在明确表
明只能添加符合模式的输入。

您现在可以更改parse以获取明确的模式描述,并在必要的地方添加一个洞来将String输入转换为
SchemaType 模式。此处显示了类型检查的parse的新定义。

清单 6.17 更新parseCommand使其解析符合模式的输入

```
parseCommand : (schema : Schema) -> String -> String -> Maybe (Command schema) parseCommand schema add rest = Just
(Add (?parseBySchema rest)) parseCommand schema get val = case all isDigit (unpack val) of
```

错误 => 没有
True => Just (Get (cast val))

parseCommand 模式 quit = 刚退出

parseCommand --- = 没有

添加一个用于转换的孔
字符串输入到所需
SchemaType 方案

→ 解析: (架构:架构) ->

(输入:字符串) -> 也许 (命令模式)

解析模式输入 = case span (/=) 输入

(cmd, args) => parseCommand 模式 cmd (ltrim args)

添加可以传递给 parseCommand 的显式模式参数以告诉它有
效输入的形式

生成的空洞有一个类型,说明它将String转换为SchemaType 模式的适当实例:

```
时间表:时间表
休息:字符串
```

```
parseBySchema : 字符串 -> SchemaType 模式
```

不过这里还是有问题!这个函数不能是全部的,因为不是每个String都可以作为模式的有效实例进行解析。
尽管如此,您目前的目标只是再次对整个程序进行类型检查。我们很快就会回到这个问题。

您已经改进了Command 的类型,将schema参数添加到parse,并插入了用于显示条目(?display)和将
用户输入转换为条目(?parseBySchema) 的孔。剩下的就是更新processInput和main以使用新定义。这些
是微小的变化,显示在下一个清单中。在processInput 中,您将当前模式传递给解析,并在main中将初始模
式设置为简单地接受字符串。

清单 6.18 使用默认模式更新了 processInput 和 main

添加一个额外的参数来解析,以便它知道使用哪个模式来解析数据

```
processInput:数据存储->字符串->可能 (字符串,数据存储)
processInput 存储输入
  = case parse (schema store) 输入
    Nothing => Just ( Invalid command\n , store)
    只是 (添加项目)->
      只是 ( “身份证” ++ show (size store) ++ \n , addToStore store item)
    Just (Get pos) => getEntry pos store
    只是退出 => 没有
  主要:我 ()
main = replWith (MkData SString
  - []
  "命令：“ processInput
```

因为 item 具有 SchemaType 类型 (模式存储)
在这里,像以前一样调用 addToStore 就可以了。

将初始架构设置为 SString,仅表示字符串。我们将很快为用户添加一种设置模式的方法。

回顾一下,您更新了 DataStore 类型以允许用户定义模式,定义它使用记录来免费获取字段访问功能,并且您已更新程序的其余部分,以便它现在进行类型检查,为零件临时插入孔
更难立即纠正。

6.3.4 在商店中显示条目

在执行此程序之前,您现在需要填补两个漏洞。第一个是 ?display,它将存储中的条目转换为字符串,其中的架构 store 给出了数据的形式:

```
存储:数据存储
id : 鳌 (大小商店)
位置:整数
store_items : Vect (大小存储) (SchemaType (模式存储))
```

显示:SchemaType (模式存储) -> 字符串

在这种情况下,使用 Ctrl-Alt-L 将孔提升到顶级函数会给您很多不需要实现显示的信息。你真正需要的是一个架构描述和数据。

相反,您可以手动实现显示,如下所示:

1类型 首先,您可以编写比 Idris 建议的更通用的类型。而不是特别是使用从商店中提取的模式,您可以根据任何模式显示数据:

显示:SchemaType 模式 -> 字符串

2定义 为了定义这个函数,你需要了解架构本身。否则,您将不知道数据的格式。添加一个骨架定义,然后将隐式参数模式带入范围:

```
显示:SchemaType 模式 -> 字符串
显示 {schema} 项目 = ?display_rhs
```

3定义 - 您可以通过模式上的大小写拆分来定义函数。因为项目类型是SchemaType 模式,模式上的案例拆分会给你更多有关预期项目类型的信息：

```
显示:SchemaType 模式 -> 字符串
显示 {schema = SString} 项目 = ?display_rhs_1
显示 {schema = SInt} 项目 = ?display_rhs_2
显示 {schema = (x,+,y)} 项目 = ?display_rhs_3
```

4类型 - 检查每个生成的孔(?display_rhs_1, ?display_rhs_2和?display_rhs_3)告诉您在每种情况下必须包含哪些项目：

```
项目:字符串
-----
display_rhs_1 : 字符串
项目:诠释
-----
display_rhs_2 : 字符串
x:架构
y:架构
项目: (架构类型 x, 架构类型 y)
-----
display_rhs_3 : 字符串
```

5细化 对于?display_rhs_1和?display_rhs_2 ,您可以通过将项目直接转换为字符串来完成定义。对于?display_rhs_3,您可以对项目进行大小写拆分并递归显示每个条目：

```
显示:SchemaType 模式 -> 字符串
显示 {schema = SString} 项目 = 显示项目
显示 {schema = SInt} 项目 = 显示项目
显示 {schema = (x,+,y)} (iteml, itemr)
    = 显示项目 ++
        ++
    显示项目
```

完成此定义并将文件重新加载到 Idris REPL 中后，
应该是剩下的一个洞， ?parseBySchema。

6.3.5 根据模式解析条目

剩下的洞， ?parseBySchema,旨在将字符串转换为用户
为模式输入了适当的类型。您可以通过查看其类型来了解预期内容：

```
休息:字符串
时间表:时间表
-----
parseBySchema : 字符串 -> SchemaType 模式
```

正如您之前看到的,并非每个String都会导致有效的SchemaType 模式,因此您可以稍微改进这种类型并创建一个返回Maybe (Schema
键入 schema)以反映解析输入可能失败的事实,另外使该方案明确：

```
parseBySchema : (schema : Schema) -> String -> Maybe (SchemaType schema)
```

然后,您可以编辑parseCommand以使用此新功能。如果parseBySchema失败(即返回Nothing) , parseCommand也应该返回Nothing:

```
parseCommand schema add rest = case parseBySchema schema rest of
    没有 => 没有
    只是补货 => 只是 (添加补货)
```

要解析模式描述的完整输入,您需要能够根据模式的子集解析输入的一部分。例如,给定一个模式

(SInt .+ SString)和输入100 Antelopes ,您需要能够解析前缀100作为SInt,然后是余数,“羚羊”,作为SString。

因此,您可以使用以下两个组件定义您的解析器: parsePrefix根据模式读取输入的前缀并返回

解析的输入(如果成功)与文本的其余部分配对。

parseBySchema使用一些输入调用parsePrefix并确保一旦它有根据模式解析输入,没有输入剩余。

下一个清单显示了parseBySchema的顶级实现和类型

parsePrefix辅助函数。

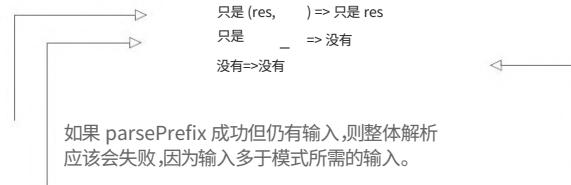
清单 6.19 parseBySchema 的概要实现,使用未定义的 parsePrefix函数根据模式解析输入的前缀

```
parsePrefix : (schema : Schema) -> String -> Maybe (SchemaType schema, String)
```

```
parseBySchema : (schema : Schema) -> String -> Maybe (SchemaType schema)
```

parseBySchema 模式输入 = case parsePrefix 模式输入或

解析成功时
parsePrefix 成功并且
输入的其余部分为空。



parsePrefix 失败,
因此整体解析应该
失败。

我们来看看parsePrefix 的大纲,遵循 type-define-refine

接近并查看模式的结构如何为我们提供有关如何进行的提示

对于实现的每个部分:

1定义 你已经有了类型,所以首先提供一个骨架定义:

```
parsePrefix : (schema : Schema) -> 字符串 ->
    也许 (SchemaType 模式,字符串)
parsePrefix 模式项 = ?parsePrefix_rhs
```

2定义 如果您在模式上进行案例拆分, Idris 将为每个可能的情况生成案例
模式的形式:

```
parsePrefix : (schema : Schema) -> 字符串 ->
    也许 (SchemaType 模式,字符串)
parsePrefix SString 输入 = ?parsePrefix_rhs_1
```

```
parsePrefix SInt 输入 = ?parsePrefix_rhs_2 parsePrefix (x .+. y) 输入 = ?
parsePrefix_rhs_3
```

- 3类型** 查看漏洞的类型可以告诉您每种模式的返回类型必须是什么。例如,在?parsePrefix_rhs_2中,如果可能,您需要将输入转换为Int,并与输入的其余部分配对:

```
输入:字符串
-----
parsePrefix_rhs_2 : 也许 (Int, String)
```

- 4 Refine -**对于?parsePrefix_rhs_2,如果输入的前缀包含数字,您可以将它们转换为Int并返回结果Int和String的其余部分。您可以将其细化为以下内容:

```
parsePrefix SInt input = case span isDigit input of (    , rest) => Nothing (num, rest) => Just
                                         (cast num, ltrim rest)
```

如果包含数字的输入的前缀为空,则它不是有效的Int,因此返回Nothing。否则,您可以将前缀转换为Int并返回输入的其余部分,并使用ltrim修剪前导空格。

- 5精炼**您可以类似地精炼?parsePrefix_rhs_1,寻找一个左引号,然后读取字符串,直到到达右引号。您很快就会在清单 6.20 中看到完整的定义。

- 6 Refine -**?parsePrefix_rhs_3的类型表明您需要解析两个子模式并组合结果:

```
x:架构
y:模式输入:字符串
-----
parsePrefix_rhs_3 : 也许 ((SchemaType x, SchemaType y), String)
```

在继续之前给x和y更有意义的名字是个好主意:

```
parsePrefix (schemal .+. schemar) 输入 = ?parsePrefix_rhs_3
```

要细化parsePrefix_rhs_3,您可以根据schemar递归解析输入的第一部分,如果成功,则根据schemar解析输入的其余部分。解析第一部分:

```
parsePrefix (schemal .+. schemar) 输入 = case parsePrefix 模式输入
                                         什么都没有 => 什么都没有 只
                                         是 (l_val, input ) => ?parsePrefix_rhs_2
```

如果模式第一部分的parsePrefix失败,整个事情都会失败。否则,你有一个新的洞:

```
架构:架构
l_val:SchemaType 模式输入 :字符串
```

```

架构师:架构
输入:字符串
-----
parsePrefix_rhs_2 : 也许 ((SchemaType schemal, SchemaType schemar), String)

```

继续重新加载!在遵循这种类型驱动的方法时,您总是有一个尽可能进行类型检查的文件。在这里,您没有完全填满这个洞,而是用一个新洞写了一小部分,并在继续之前检查了您的类型检查。

7 Refine 最后,您可以通过根据schemar 解析剩余的输入来完成此案例:

```

parsePrefix (schemal .+. schemar) 输入 = case parsePrefix 模式输入

没有=>没有
Just (l_val, input ) => case parsePrefix
    schemar input 的
    没有=>没有
    只是 (r_val, input ) =>
        只是 ((l_val, r_val), 输入 )

```

嵌套案例块 我们使用的这些嵌套案例块可能看起来有点冗长。在 6.3.7 节中,您将看到一种更简洁的编写方式。

下面的清单显示了parsePrefix 的完整实现,填写了剩余的细节,包括解析带引号的字符串。您现在拥有一个可以编译和运行的完整实现。

清单 6.20 根据特定模式解析输入的前缀

将输入的前缀解析为带引号的字符串。如果输入不是以引号字符开头,则解析应该失败。

getQuoted 返回带引号的字符串前缀,
输入分解为字符作为列表字符。

```
parsePrefix : (schema : Schema) -> String -> Maybe (SchemaType schema, String) parsePrefix SString input = getQuoted (unpack input)
```

在哪里

```
getQuoted : List Char -> Maybe (String, String) getQuoted (      :: xs) = case span (/=)
    xs of (quoted,
```

```
        :: rest) => Just (pack 引用, ltrim (pack rest))
```

得到报价
— => 没有
— = 没有

```
parsePrefix SInt input = case span isDigit input of (      , rest) => Nothing (num, rest) => Just
    (cast num, ltrim rest)
```

使用 ltrim 从输入的其余部分中删除任何前导空格。

将输入的前缀解析为整数:它采用完全由数字组成的字符串的前缀。如果没有数字,解析应该会失败。

```

parsePrefix (schema .+. schemar) 输入
= case parsePrefix 模式输入
    没有=>没有
    只是 (l_val, input ) =>
        case parsePrefix schemar 输入 的
            没有=>没有
            只是 (r_val, input ) => 只是 ((l_val, r_val), input )

```

根据 schemal 解析字符串的前缀,然后根据 schemar 解析字符串的其余部分。如果要么部分失败,整体解析应该失败。

6.3.6 更新模式

尽管您现在有了完整的实现,但它仍然没有更多功能与之前的版本相比,因为架构在 main 中被初始化为 SString,并且你还没有实现任何更新它的方法:

```

主要:我 ()
main = replWith (MkData SString
    _ [])) 命令: 进程输入

```

您至少可以通过更新 main 和重新编译来尝试不同的模式。为了例如,您可以尝试接受两个字符串和一个 Int 的模式:

```

主要:我 ()
main = replWith (MkData (SString .+. SString .+. SInt)
    "命令: " processInput
    _ [])

```

您可以在 REPL 中使用:exec 编译和运行它,并尝试几个示例条目:

```

*数据存储> :exec
命令:添加 "Bob Dylan" "Blonde on Blonde" 1965
编号 0
命令:添加 "Prefab Sprout" "从兰利公园到孟菲斯" 1988
编号 1
命令:获取 0
"鲍勃·迪伦", "金发女郎", 1965

```

然而,最好允许用户定义他们自己的模式,而不是而不是将它们硬编码到 main 中。为此,您可以添加一个新命令来设置新模式,更新命令数据类型。

清单 6.21 带有用于更新模式的新命令的 Command 数据结构

```

数据命令:架构 -> 输入位置
    SetSchema : (newschema : Schema) -> 命令模式
    添加:SchemaType 模式 -> 命令模式
    获取:整数 -> 命令模式
    退出:命令模式

```

用于设置新模式的新命令。请注意,该类型表示新闻模式和现有模式之间没有关系。

您还需要执行以下操作的函数:

更新 DataStore 类型以保存新模式。这应该只工作当商店为空时,因为当您更改模式类型时,它会使商店的当前内容无效。

从用户输入中解析模式描述。

您还需要更新parseCommand和processInput来处理解析和
处理新命令。这些新功能使用相同的实现
到目前为止,您在实施扩展数据存储时所遵循的过程。清单 6.22
显示解析新命令的工作原理。它添加一个用户命令模式,后跟一个字符串和整数列表,并将其转换为
SetSchema命令。

**清单 6.22 解析Schema描述,并将Command的解析器扩展为
支持设置新架构**

```

parseSchema : 列表字符串 -> 可能模式
parseSchema ( 字符串 :: xs)
    = 案例 xs 的
        [] => 只是 SString
        _  => case parseSchema xs of
            没有=>没有
            只是 xs_sch => 只是 (SString .+ xs_sch)

parseSchema ( Int :: xs)
    = 案例 xs 的
        [] => 只是 SInt
        _  => case parseSchema xs of
            没有=>没有
            只是 xs_sch => 只是 (SInt .+ xs_sch)

解析模式
    = 没有

parseCommand : (schema : Schema) -> String -> String -> Maybe (Command schema)
{ ... 其余定义如前 ... }

parseCommand 模式 “模式”休息
    = case parseSchema (words rest) of
        没有=>没有
        只是 schemaok => 只是 (SetSchema schemaok)

解析命令
    --- = 没有

```

清单 6.23 展示了在新命令执行后如何更新模式
解析。作为一种设计选择,它只允许在存储时更新模式
为空,因为没有更新数据存储内容的通用方法
使用任意更新的模式(另一种方法是在
用户更改架构)。

**清单 6.23 处理SetSchema命令,更新模式描述
在数据存储中**

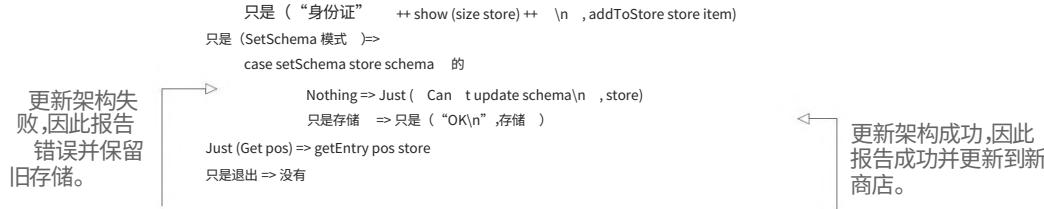
```

setSchema : (store : DataStore) -> Schema -> Maybe DataStore
setSchema store schema = case size store of
    Z => Just (MkData 模式)
    S k => 没有

processInput 数据存储->字符串->可能 (字符串,数据存储)
processInput 存储输入
    = case parse (schema store) 输入
        Nothing => Just ( Invalid command\n , store)
        只是 (添加项目)->

```

使用模式增强交互式数据存储



最后,您可以编译并运行该程序,并尝试从用户输入中设置新模式:

```

*数据存储> :exec
命令:模式整数字符串
好的
命令:添加 99 个 “红气球”
编号 0
命令:添加 76 个 “长号”
编号 1
命令:schema String String Int
商店中的条目时无法更新架构
命令:得到 1
76、《长号》

```

最后,使用数据类型来描述模式并使用该模式来计算对数据存储的操作类型会产生一些后果:

在读取用户输入时,如果根据模式无效,则不能将输入添加到存储中。如果更改模式类型,则不能使存储的内容无效,因为存储内容的类型会阻止它。更改架构类型需要您有一个空存储。在为用户输入编写解析器时,您可以使用模式的描述来指导解析器的实现并为输入构建正确的类型。

6.3.7 使用 do 表示法使用 Maybe 对表达式进行排序

在数据存储程序中,有几个地方我们使用了case块来检查一个函数的结果,该函数返回一个Maybe类型的東西,并将该结果传递出去。例如:

```

parseCommand 模式 “模式”休息
= case parseSchema (words rest) of
    没有=>没有
    只是 schemaok => 只是 (SetSchema schemaok)

```

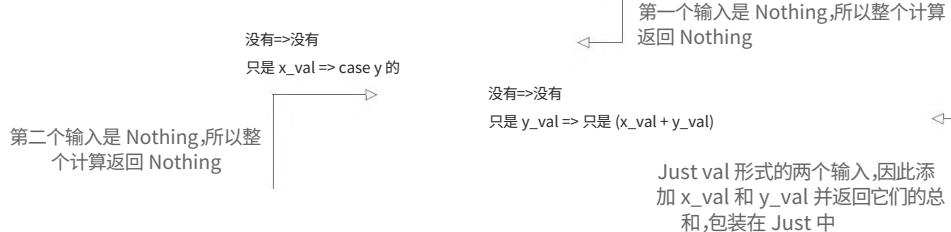
在这里,返回 Maybe类型 (命令模式)的parseCommand调用了 parseSchema,它返回了Maybe Schema 类型的東西,并使用case表达式检查调用parseSchema 的结果。

如果parseSchema失败，则parseCommand也会失败。同样，如果parseSchema成功，则parseCommand也成功。

您可以在以下清单中的maybeAdd函数中看到类似的模式，它添加了两个Maybe Int类型的值，如果任一输入为Nothing，则返回Nothing。

清单 6.24 添加两个Maybe Int (Maybe.idr)

```
MaybeAdd : Maybe Int -> Maybe Int -> Maybe Int
maybeAdd x y = case x of
```



您可以在一些示例上尝试maybeAdd，您会看到如果两者的形式都是Just val的形式，它会添加其输入以获得一些具体的值val，并且如果其中一个输入是Nothing，它会返回Nothing：

```
*Maybe> MaybeAdd (Just 3) (Just 4)
Just 7 : 也许是 Int
```

```
*也许>也许添加 (仅 3 个)无
什么都没有:也许是 Int
```

```
*也许>也许什么都不加 (只有 4 个)
什么都没有:也许是 Int
```

在这里、在parseCommand以及整个数据存储实现中的其他几个地方都可以找到一个常见的模式：

1计算类型为Maybe ty 的表达式。结果是Nothing或Just x，

其中x的类型为ty。

2如果结果为Nothing，则整个计算的结果为Nothing。

3如果结果是Just x，则将x传递给计算的其余部分。

当您看到一个常见模式时，最好尝试在高阶函数中捕获该模式。事实上，你已经在第 5 章中看到了一个实现类似模式的操作符，用于对IO操作进行排序：

```
(>>=): I a -> (a -> I b) -> I b
```

相同的运算符适用于排序Maybe计算，定义如下：

```
(>>=) : 也许 a -> (a -> 也许 b) -> 也许 b
(>>=) 没有下一个 = 没有
(>>=) (只是 x) 下一个 = 下一个 x
```

实际上,如果成功(即返回一个Just),它将获取第一个计算的输出,并将其作为输入传递给第二个。它捕捉了一个常见的模式
使用Maybe类型评估表达式。

(>=) 的类型

请记住,从第5章开始,如果您在REPL中检查(>=)的类型,您会看到受约束的泛型类型:

```
伊德里斯> :t (>=)
(>=) : Monad m => ma -> (a -> mb) -> mb
```

通常,>=运算符可用于对计算进行排序。你会看到如何这在第7章讨论接口时起作用。

使用(>=)作为中缀运算符,您可以更简洁地重写MaybeAdd,如果有些神秘的话,如下面的清单所示。

清单 6.25 添加两个Maybe Ints 使用(>=)而不是直接案例分析

```
MaybeAdd : Maybe Int -> Maybe Int -> Maybe Int
也许添加 xy = x >= \x_val =>
如果 Idris 评估 y >= \y_val =>
>= 的第二个操作数
在这里,你必须有
值只是 y_val.
```

如果 Idris 在此处计算 >= 的第二个操作数,则
>= 的定义意味着 x 必须具有值 Just x_val。

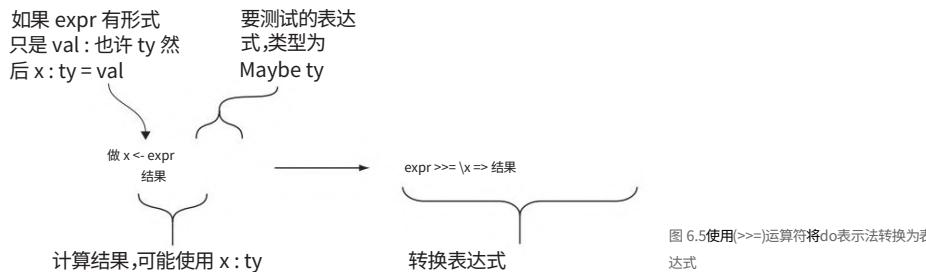
只是 (x_val + y_val)

在第5章中,您看到Idris提供了一种特殊的表示法,用于使用(>=)进行排序计算,由关键字do引入。您可以在那里使用相同的符号,
并写出maybeAdd如下。

清单 6.26 使用do表示法而不是使用(>=)添加两个Maybe Int 直接地

```
MaybeAdd : Maybe Int -> Maybe Int -> Maybe Int
可能添加 xy = do x_val <- x
如果 x 的形式为 Just x_val,则继
续计算;否则,它返回 Nothing.
y_val <- y
只是 (x_val + y_val)
如果 y 的形式为 Just y_val,则继
续计算;否则,它返回 Nothing.
```

图6.5显示了如何将使用do表示法的表达式转换为表达式使用(>=)。这与翻译编写的IO程序的工作方式完全相同
使用do表示法。就像将列表语法翻译成Nil和(:)一样,这种翻译是纯语法的。因此,如果您在其他上下文中定义(>=)
运算符,Idris将允许您在该上下文中使用do表示法。



使用do表示法,您可以为parseCommand编写“模式”案例,如下所示,让do表示法通过($>>=$)处理Nothing,这样您就可以专注于Just schemaok的成功案例:

```
parseCommand schema schema rest = do schemaok <-
    parseSchema (words rest)
    Just (SetSchema schemaok)
```

练习

1 更新数据存储程序以支持模式中的字符。

您可以在REPL上测试您的解决方案,如下所示:

```
*ex_6_3> 执行
命令: 模式 Char Int
好的
命令: 加 x 24
编号 0
命令: 添加 y 17
编号 1
命令: 获取 0 x ,24
```

2 修改get命令,如果没有给出参数,它会打印整个con
数据存储的帐篷。

例如:

```
*ex_6_3> 执行
命令: 模式 Char Int
好的
命令: 加 x 24
编号 0
命令: 添加 y 17
编号 1
命令: 获取 0: x ,24
1: y ,17
```

3 更新数据存储程序,使其在适当的情况下使用do表示法而不是嵌套的case块。

6.4 总结

类型同义词是现有类型的替代名称,它允许您给出
类型的更具描述性的名称。

类型级函数是可以在 Idris 期望类型的任何地方使用的函数。类型同义词是类型级函数的一种特殊情
况。类型级函数可用于计算类型,它们允许您编写

具有不同数量参数的函数,例如printf。

您可以将数据存储的模式表示为一种类型,然后使用类型级函数来计算操作的类型,例如
从模式描述中解析和显示数据存储中的条目。

记录是一种数据类型,只有一个构造函数,并具有自动生成的函数,用于从记录中投影字
段。提炼数据类型后,可以使用孔来纠正编译错误

暂时。

使用do表示法,您可以对使用Maybe的计算进行排序,以捕捉错误的可能性。

接口：使用受约束的泛型类型

本章涵盖

使用接口约束泛型为特定上下文实现接口使用
Prelude 中定义的接口

在第 2 章中，您看到可以限制具有类型变量的泛型函数类型，以便变量代表更有限的类型集。例如，您看到以下函数将任何数字类型的数字加倍：

```
double : 数字 a => a -> a
双倍=x+x
```

double 的类型包括一个约束 Num a，它表明 a 只能代表数字类型。因此，您可以将 double 与 Int、Nat、Integer 或任何其他数字类型一起使用，但如果尝试将其与非数字类型（例如 Bool）一起使用，Idris 将报告错误。

您已经看到了一些这样的约束，例如支持相等测试的类型的 Eq 和支持比较的类型的 Ord。您还看到了以下功能

依赖其他我们没有详细讨论的约束,比如map和 $>=$,它们分别依赖于Functor和Monad。我们还没有讨论这些如何定义或引入约束。

在本章中,我们将讨论如何定义和使用受约束的泛型类型接口。例如,在double的类型声明中,约束Num a是由一个接口Num实现,该接口描述了将要进行的算术运算针对不同的数值类型以不同的方式实现。

Haskell 中的类型类如果你了解 Haskell,你就会熟悉 Haskell 的类型类概念。Idris 中的接口类似于 Haskell 和经常以相同的方式使用,尽管有一些不同。最重要的是,首先,Idris 中的接口可以由任何类型的值参数化,并且不限于类型或类型构造函数,

其次,Idris 中的接口可以有多种实现,不过我们不会在本章中详细介绍。

从类型驱动开发的角度来看,接口允许我们为泛型类型提供必要的精度级别。一般来说,Idris 中的接口描述了一个可以针对不同具体类型以不同方式实现的通用操作的集合。例如:

您可以为序列化和加载泛型的操作定义一个接口
传入和传出 JSON 的数据。

图形库可以为绘图表示的操作提供接口
的数据。

Prelude 包含多种接口,我们将在本文中重点介绍这些章节。我们将首先详细研究其中最重要的两个。

7.1 与 Eq 和 Ord 的一般比较

首先,我们将看两个定义泛型比较的接口,这两个接口在 Prelude 中定义:

Eq,支持比较值是否相等。

Ord,支持比较值以确定哪个更大。

在此过程中,您将学习如何声明接口,如何实现这些接口在特定的上下文中,以及不同的接口如何相互关联。

7.1.1 用 Eq 测试相等性

正如你在第 2 章中看到的,Idris 提供了一个运算符来测试值是否相等, $=$,具有受约束的泛型类型:

伊德里斯> :t (=)

(=): 等于你 => 你 -> 你 -> 布尔

换句话说，你可以比较一些泛型类型ty的两个值是否相等，如果ty满足Eq约束，则将结果作为Bool返回。相似地，有一个用于测试不等式的运算符：

```
伊德里斯> :t (/=)
(/=): Eq ty => you -> you -> Bool
```

指定多个约束

在本节的示例中，我只指定了一个约束，即 Eq ty。你可以还将多个约束列为逗号分隔的列表。例如，以下函数类型指定 ty 需要满足 Num 和 Show 约束：

```
addAndShow : (Num ty, Show ty) => ty -> ty -> String
```

稍后您将在第 7.2.2 节中看到更多示例。

任何时候使用这些运算符中的任何一个，都必须知道它的操作数是可以比较相等的类型。比方说，例如，你想写一个计算特定值的出现次数的函数，一些通用的在列表中键入ty。我们将在文件 Eq.idr 中创建一个名为occurrences的函数：

1 **类型** 与往常一样，首先为函数指定类型并创建骨架
定义：

```
出现次数: (项目:ty) > (值:列表 ty) > Nat
出现项 xs = ?occurrences_rhs
```

2 **定义、细化** 您可以通过对输入列表xs进行大小写拆分来定义函数，并细化每种情况下产生的孔。如果它是一个空列表，可以没有出现项目，所以返回0：

```
出现次数: (项目:ty) > (值:列表 ty) > Nat
出现项 [] = 0
出现项 (值::值)= ?occurrences_rhs_2
```

3 **Refine** 如果输入是非空列表，请尝试在列表的开头测试值相等的列表和项目：

```
出现次数: (项目:ty) > (值:列表 ty) > Nat
出现项 [] = 0
出现项 (值::值)= 案例值==项目
case_val => ?occurrences_rhs_2
```

不幸的是，Idris 报错：

```
eq.idr:3:13:
在使用预期类型检查事件的右侧时
纳特
```

找不到 Eq ty 的实现

这个问题和我们在第三章定义的时候遇到的类似
`ins_sort`。在这种情况下，`value`和`item`的类型为`ty`，但没有约束
`ty`类型的值对于相等性是可比较的。

4细化 解决方案，与`ins_sort`一样，是通过添加约束来细化类型：

出现次数`:Eq ty => (项目:ty) -> (值:列表 ty) -> Nat`

约束`Eq ty`意味着您现在可以使用`==`运算符并完成
 定义如下：

出现次数`:Eq ty => (项目:ty) -> (值:列表 ty) -> Nat`

出现项`[] = 0`

出现项目`(值:值)=案例值==项目`

`False => 出现项值`

`True => 1 + 出现次数项值`

最后一种类型表示它需要一个`ty`和一个`List ty`作为输入，前提是
`ty`类型变量代表可以比较是否相等的类型。

这适用于内置类型，例如`Char`和`Integer`：

`*Eq> 出现 b [a , a , b , b , b , c]`
 3.纳特

`*Eq> 出现 100 [50,100,100,150]`
 2.纳特

但是如果你有用户定义的类型呢？假设您有一个名为的用户定义类型
`Matter`，也在`Eq.idr`中定义：

数据`Matter = 固体 | 液体 | 气体`

您希望能够计算列表中`Liquid`的出现次数。不幸的是，如果你尝试这个，`Idris`会报告一个错误：

`*Eq> 出现 Liquid [Solid, Liquid, Liquid, Gas]`
 找不到`Eq Matter`的实现

此错误消息意味着`Idris`不知道如何比较用户定义的`Matter`类型中的值是否相等。为了纠正这个问题，我们需要看看如何

`Eq`约束是如何定义的，以及如何向`Idris`解释用户定义的类型如
`Matter`可以满足它。

7.1.2 使用接口和实现定义 Eq 约束

诸如`Eq`之类的约束在`Idris`中使用接口定义。接口定义
 包含相关函数（称为接口的方法）的集合，可以
 给定针对特定上下文的不同实现。`Eq`接口定义在
`Prelude`，包含方法`(==)`和`(/=)`。

清单 7.1 Eq接口（在 Prelude 中定义）

接口声明引入了可用于约束泛型函数的新接口。

→ 接口 Equity where
 $(==) : ty \rightarrow ty \rightarrow \text{布尔}$
 $(/=) : ty \rightarrow ty \rightarrow \text{布尔}$

方法声明介绍
 必须由接口的实现定义的功能。

定义接口引入了新的顶级函数
 对于接口的每个方法。清单 7.1 中的接口声明引入了顶层函数 $(==)$

和 $(/=)$ 。如果您检查这些函数的类型，
 你会在他们的类型中看到明确的 Eq ty 约束：

$(==) : \text{等你} \Rightarrow you \rightarrow you \rightarrow \text{布尔}$
 $(/=) : Eq\ ty \Rightarrow you \rightarrow you \rightarrow \text{Bool}$

图 7.1 显示了接口声明的组成部分。在这种情况下，参数 ty 被假定为

type 默认情况下是类型，代表要被约束的泛型参数。参数必须

出现在每个方法声明中。

参数命名约定 接口的参数（此处为 ty ）
 通常是泛型类型变量，所以我通常给它们简短、泛型、
 名字。名称 ty 表示参数可以是任何类型。什么时候
 有更多的参数，或者接口有更具体的用途，我给出更具体的名称。

然后，接口声明为相关函数提供类型，其中的参数代表通用参数。要为特定情况定义这些函数，您

需要提供实现。

要编写实现，您需要提供接口和参数，以及
 每个方法的定义。以下清单显示了一个实现
 对于 Eq 接口，它解释了 Matter 类型如何满足约束。

清单 7.2 Eq for Matter (Eq.idr) 的实现

情商在哪里
 $(==) \text{ 实心实心} = \text{真}$
 $(==) \text{ 液体液体} = \text{真}$
 $(==) \text{ 气体气体} = \text{真}$
 $(==) \text{ } \dots = \text{假}$

一个实现声明，这里解释了如何满足 Matter 类型的 Eq 约束

Matter 的 $(==)$ 方法的实现

$(/=) \text{ xy} = \text{不} (x == y)$

Matter 的 $(/=)$ 方法的实现



图 7.1 Eq 接口声明

实现的交互式开发正如您将很快看到的,您可以使用 Ctrl-Alt-A 来提供Eq Matter 的框架实现,就像函数定义一样。

如果将此实现添加到定义了出现次数的文件中,并且
重要的是,您现在可以将事件与List Matter 一起使用:

*Eq> 出现 Liquid [Solid, Liquid, Liquid, Gas]
2.纳特

这提供了两种方法的实现, (==)和(/=),为了能够比较Matter类型的元素是否相等和不相等,需要实现这两种方法。请注意,实现不包含任何类型声明,因为这些是在接口声明中给出的。

[接口文档](#)如果您在 REPL 的接口上

使用 :doc,或在 Atom 中的接口名称上使用 Ctrl-Alt-D,Idris 将显示该接口的文档,包括其参数、方法和已知实现的列表。例如,下面是 Eq 的样子:

```
伊德里斯> :doc 方程
接口方程
Eq 接口定义了不等式和等式。

参数:
泰

方法:
(==): 等于你 => 你 -> 你 -> 布尔
    中缀 5
    函数是总计
(/ =): Eq ty => you -> you -> Bool
    中缀 5
    函数是总计
实现 Eq ()
Eq Int Eq
Integer Eq
Double [...]
```

您可以在 Atom 中以交互方式构建实现,如下所示:

1种类型 首先单独给出实现标头:

情商在哪里

这是一个 Type 步骤，因为它使用 Matter 实例化(==)和(/=)的类型代表参数 a。

2 定义 您还可以使用 Atom 中的交互式编辑工具来构建接口的实现。将光标放在 Eq 上，按 Ctrl-Alt-A，然后按 Idris 将为(==)和(/=)添加骨架定义，并为论据：

情商在哪里

```
(==) xy = ?Eq_rhs_1
(/=) xy = ?Eq_rhs_2
```

3 类型 - 如果您检查 ?Eq_rhs_1 的类型，您将看到确认 x 和 y 属于物质类型：

x : 物质

y : 问题

Eq_rhs_1 : 布尔

4 定义 您可以通过在 x 上区分大小写来定义(==)：

情商在哪里

```
(==) 实心 y = ?Eq_rhs_3
(==) 液体 y = ?Eq_rhs_4
(==) 气体 y = ?Eq_rhs_5
```

```
(/=) xy = ?Eq_rhs_2
```

5 精炼 每个 Matter 类型的值都只等于它自己，所以你可以精炼定义如下，当输入不是时，有一个包罗万象的情况来处理平等的：

情商在哪里

```
(==) 实心 实心 = 真
(==) 液体 液体 = 真
(==) 气体 气体 = 真
(==) _ _ = 假
```

```
(/=) xy = ?Eq_rhs_2
```

请记住，案例的顺序很重要，伊德里斯会尝试匹配子句按顺序排列，所以包罗万象的情况必须在最后。

6 精炼 要完成实施，您需要定义(/=)。模拟人生，请定义是使用(==)然后反转结果：

情商在哪里

```
(==) 实心 实心 = 真
(==) 液体 液体 = 真
(==) 气体 气体 = 真
(==) _ _ = 假
```

```
(/=) xy = 不 (x == y)
```

与 Eq 和 Ord 的一般比较

当你声明一个接口时,你引入了一个相关的新泛型集合函数,称为方法,可以针对特定情况进行重载。当你定义接口的实现,您必须为其所有定义方法。因此,当您为 Matter 定义 Eq 实例时,您必须同时提供(==)和(/=) 的定义。

7.1.3 默认方法定义

接口定义了相关方法的集合,例如(==)和(/=)。在一些在这种情况下,方法密切相关,您可以根据接口中的其他方法来定义它们。例如,您总是希望 $x \neq y$ 的结果是

与 $x == y$ 的结果相反,无论 x 和 y 的值甚至类型是什么。

对于这种情况,Idris 允许您为方法提供默认定义。如果实现没有为具有默认定义的方法提供定义,Idris 使用该默认值。例如, Eq 接口为两者提供默认值

(==) 和 (/=),每一个都根据另一个定义,如下所示。

清单 7.3 带有默认方法定义的 Eq 接口

```
接口 Eq a where
  (==) : a -> a -> Bool
  (/=) : a -> a -> Bool
  (==) xy = 不 (x /= y)
  (/=) xy = 不 (x == y)
```

默认方法实例,如果方法不存在于接口的特定实现中时使用

因此,您可以通过仅提供(==) 的定义,并使用(/=) 的默认方法实现:

情商在哪里
 (==) 实心实心 = 真
 (==) 液体液体 = 真
 (==) 气体气体 = 真
 (==) _ _ = 假

默认方法定义意味着当您定义实现时,对于 Eq, 您可以为(==) 和 (/=) 中的一个或两个提供定义。

7.1.4 约束实现

在为泛型类型编写实现时,您可能会发现需要对泛型类型的参数进行附加约束。例如,要检查

两个列表是否相等,您需要知道如何比较元素类型平等。

在第 4 章中,您看到了二叉树的泛型,在 tree.idr 中定义如下:

```
数据树元素 = 空
| 节点 (树元素)元素 (树元素)
```

要检查两棵树是否相等，您还需要能够比较元素类型。让我们看看如果您尝试定义Eq的实现会发生什么

具有通用元素类型的树：

1类型 首先编写实现标头，说明您使用的接口

希望实现以及您正在实现它的类型：

`Eq (树元素)在哪里`

2定义 - 添加一个骨架定义，为所有

接口方法：

`Eq (树元素)在哪里`

`(==) xy = ?Eq_rhs_1`

`(/=) xy = ?Eq_rhs_2`

您可以使用`(/=)`的默认定义，因此您可以删除第二个方法：

`Eq (树元素)在哪里`

`(==) xy = ?Eq_rhs_1`

3定义 与Matter的Eq实现一样，您可以按大小写定义`(==)`

拆分每个参数，并在输入的情况下使用包罗万象的情况

是不平等的。您知道Empty等于自身，如果所有参数都相等，则节点相等，而其他一切都不相等：

`Eq (树元素)在哪里`

`(==) 空 空 = 真`

`(==) (节点左 e 右) (节点左 e 右) = ?Eq_rhs_3`

`(==) = 假`

目前，您已经留下了一个漏洞，`?Eq_rhs_3`，用于比较的详细信息节点。

4精炼 您希望能够通过精炼

`?eq_rhs_3`孔表示可以比较每个对应的参数：

`Eq (树元素)在哪里`

`(==) 空 空 = 真`

`(==) (节点左 e 右) (节点左 e 右)`

`= 左 == 左 && e == e && 右 == 右`

`(==) = 假`

但是，不幸的是，Idris 报告了一个问题：

找不到 `Eq elem` 的实现

您可以比较`left`和`right`是否相等，并相应地比较`right`和

对，因为它们是`Tree elem`类型，并且实现可以是

递归的；您当前正在为`Tree elem`定义相等性。但是`e`和`e`是

类型`elem`，泛型类型，你不一定知道如何比较`elem`

为了平等。

与 Eq 和 Ord 的一般比较

5精炼 解决方案是精炼实现声明,将其约束为要求elem也有可用的Eq实现:

```
Eq elem => Eq (Tree elem) 其中
(==) 空空 = 真
(==) (节点左 e 右) (节点左 e 右)
      = 左 == 左 && e == e && 右 == 右
(==)      = 假
```

约束出现在箭头的左侧, \Rightarrow ,与类型声明中出现约束的方式相同。图 7.2 显示了这个接口头的组成部分。

您可以将此标题阅读为说明可以比较通用树是否相等,前提是它们的元素类型也可以比较是否相等。您可以像这样引入接口约束

如果您需要进一步约束该类型变量,则在您引入类型变量的任何地方。

实施约束

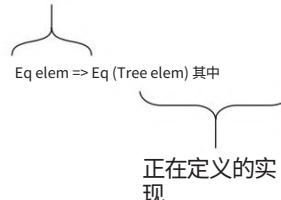


图 7.2 约束的实现头

实现参数的限制您现在已经看到了由Matter和Tree elem参数化的Eq的实现,它们都是Type类型。但是您不能通过 Type 类型的所有内容来参数化实现。您仅限于由数据或记录声明或原始类型引入的名称。特别是,这意味着您不能通过类型同义词或计算类型的函数来参数化实现。

您已经看到了类型声明的约束,在这里您已经在实现定义中看到了一个。同样,您可以对接口本身施加约束。

7.1.5 约束接口:用 Ord 定义排序

如果您对接口定义施加约束,您实际上是在扩展现有接口。例如,在 Prelude 中,有一个Ord接口,如下面的清单所示,它扩展了Eq以支持值的排序。

清单 7.4 Ord 接口,它扩展了Eq (在 Prelude 中定义)

```
接口 Eq ty => Ord ty 其中
    比较: ty -> ty -> 排序 (<): ty -> ty -> Bool (>): ty -> ty ->
    Bool (<=): ty -> ty -> Bool (>=): ty -> ty -> Bool max: ty
    -> ty -> ty min: ty -> ty -> ty
```

← 声明 Ord 接口并声明 Ord ty 的实现需要 Eq ty
的实现

序在 Prelude 中定义为 LT,EQ
或 GT。

←———— 返回两个输入中较小的一个

退货
两者中较
大的一个

输入

除了compare之外的所有方法都有默认定义，其中一些是按照compare来编写的，还有一些使用Eq接口提供的(==)方法。

如果您为某些数据类型实现了Ord，则可以对包含该类型的列表进行排序：

排序:Ord you => 给你写信 -> 给你写信

例如，您可能有一个表示音乐收藏的数据类型，其中包含标题、艺术家和发行年份的记录，您可能希望先按艺术家姓名，然后按发行年份，然后按标题对它们进行排序。下面的清单通过一些示例给出了这种数据类型的定义。

清单 7.5 一个记录数据类型和一个要排序的集合（Ord.idr）

记录专辑在哪里
构造函数 MkAlbum 艺术家:字符串标

题:字符串年份:整数
记录字段，产生艺术家、标题和年份投
影功能

← 记录声明（见第 6 章，第 6.3.2 节）

help : 专辑帮助 =
MkAlbum The Beatles Help 1965

橡胶灵魂：专辑
Rubbersoul = MkAlbum The Beatles Rubber Soul 1965

云:专辑云= MkAlbum “乔尼
米切尔”“云”1969

hunckydory : 专辑 hunckydory
= MkAlbum David Bowie Hunk Dory 1971

英雄：专辑英雄 = MkAlbum
David Bowie Heroes 1977

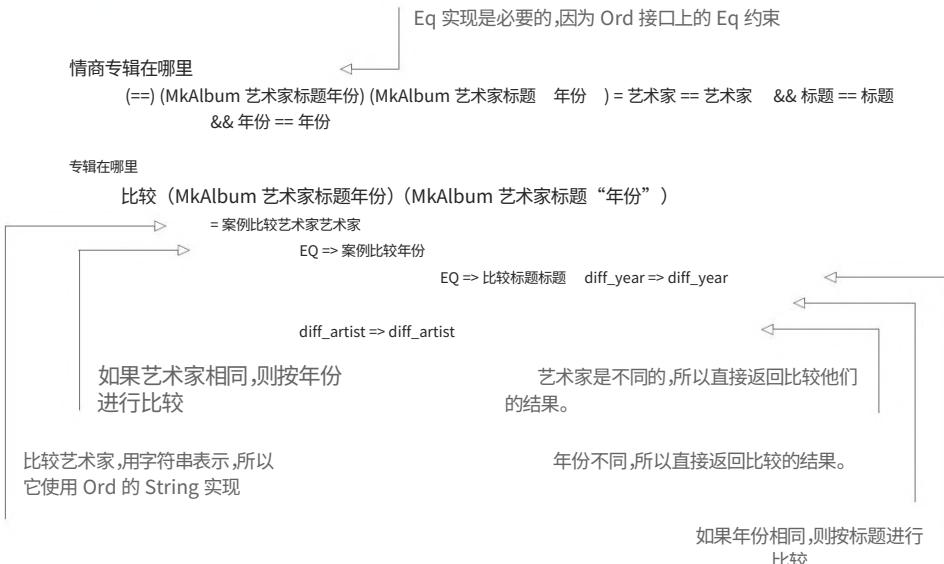
collection : List Album collection = [help,
Rubbersoul, clouds, hunckydory, heros]

如果您尝试按原样对集合进行排序，Idris 将报告它不知道如何对Album 类型的值进行排序：

*Ord> 排序集合
找不到 Ord Album 的实现

清单 7.6 展示了如何向 Idris 解释如何订购专辑。首先，你需要给出一个Eq实现，因为Ord接口的约束要求Ord的实现也是Eq的实现。 Eq实现检查每个字段是否具有相同的值； Ord实现按艺术家姓名进行比较，然后按年份进行比较，如果前两者相等，则按标题进行比较。

清单 7.6 Album 的 Eq 和 Ord 的实现



为Album实现Ord意味着您可以对 Album 类型的值使用通常的比较运算符：

*Ord>英雄>云
假:布尔值

*Ord> 帮助 <= 橡胶灵魂
真:布尔

这也意味着您可以使用任何带有Ord约束的函数,例如sort。例如,您可以对集合进行排序并按排序顺序列出标题：

*Ord> 地图标题 (排序集合)
[“Hunky Dory” , “Heroes” , “Clouds” , “Help” , “Rubber Soul”]:列表字符串

对接口进行约束,例如对Ord的Eq约束,允许您定义接口的层次结构。因此,例如,如果有某种类型的Ord实现,您知道可以安全地假设该类型也有Eq的实现。 Prelude 定义了几个接口,其中一些按层次结构排列,我们将在下一节中介绍一些最重要的接口。

练习



对于这些练习,您将使用第 4 章中定义的Shape类型：

数据形状=三角形双双

|长方形 双 双
|圆形双人间

1 实现 Eq for Shape。

您可以在REPL上测试您的答案,如下所示：

```
*ex_7_1> 圆 4 == 圆 4
真:布尔
```

```
*ex_7_1> 圆 4 == 圆 5
假:布尔值
```

```
*ex_7_1> 圆 4 == 三角形 3 2
假:布尔值
```

2 实现形状的Ord。形状应按面积排序,因此面积较大的形状被认为大于面积较小的形状。

您可以通过尝试对以下形状列表进行排序来测试这一点：

```
testShapes : 列表形状 testShapes = [Circle
3, Triangle 3 9, Rectangle 2 6, Circle 4,
矩形 2 7]
```

在REPL中对列表进行排序时,您应该看到以下内容：

```
*ex_7_1> sort testShapes [Rectangle 2.0 6.0,
Triangle 3.0 9.0, Rectangle 2.0 7.0, Circle 3.0,
Circle 4.0] 列表形状
```

7.2 Prelude 中定义的接口

正如您刚刚看到的,除了Eq和Ord之外,Idris Prelude 还提供了几个常用的接口。在本节和下一节中,我将简要介绍一些最重要的内容。我们已经遇到过几个：Show、Num、Cast、Functor和Monad。在这里,您将看到这些接口是如何定义的,它们可能在哪里使用,以及如何为您自己的类型编写它们的实现的一些示例。

接口的参数（换句话说,接口头中给出的变量）可以具有任何类型。如果接口标头中没有为参数指定显式类型,则假定其为Type 类型。在本节中,我们将看一些由Types 参数化的接口。

7.2.1 使用 Show 转换为字符串

在第 2 章中,您看到了将值转换为字符串的show函数。这是Show接口的一个方法,在 Prelude 中定义,如下表所示。 show和showPrec方法都有默认实现,每个都根据另一个定义。在这里,我们只考虑表演。

清单 7.7 Show接口（在 Prelude 中定义）

```

界面显示ty where
    显示: (x:ty) ->字符串
    showPrec : (d : Prec) -> (x: ty) -> 字符串

```

ty是接口的参数,是Type类型的变量。

将值直接转换为字符串
在优先上下文中
将值转换为字符串

优先级上下文 showPrec的目的是能够显示
如有必要,括号中的复杂表达式。如果说,这可能很有用
你有一个算术公式的表示,其中优先规则说
一些子表达式需要用括号括起来。默认情况下, showPrec
直接调用show,对于大多数显示目的来说,这完全足够了。如果
如果您想进一步调查,请使用:doc 查看文档。

Prelude 提供了两个函数,它们使用show将值转换为String和
然后将其输出到控制台,带或不带尾随换行符:

```

printLn : 显示 ty => ty -> IO ()
打印:显示 ty => ty -> IO ()

```

您可以为Album类型定义一个简单的Show实现:

```

在哪里显示专辑
演出 (MkAlbum 艺人称号年份)
    =标题++ "作者"++艺术家++
        " (发布"
        ++ 显示年份 ++
    )

```

然后,您可以使用printLn将专辑打印到控制台。例如:

```

*Ord> :exec printLn hunkydory
David Bowie 的 Hunky Dory (1971 年发行)

```

Show接口主要用于调试输出,以人类可读的形式显示复杂数据类型的值。

7.2.2 定义数字类型

Prelude 提供了具有数值操作方法的接口层次结构。

这些接口将操作员分为几组:

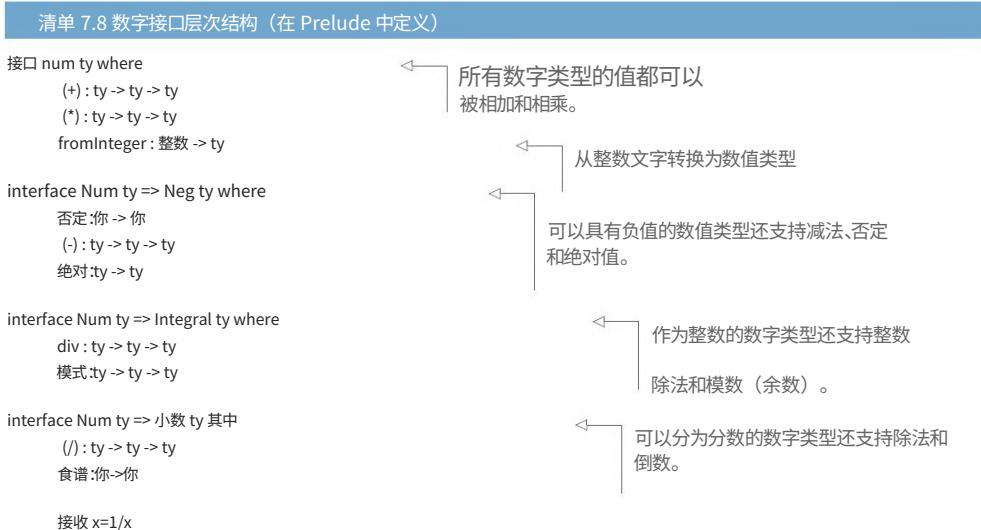
Num接口 包含适用于所有数字类型的操作,包括
整数文字的加法、乘法和转换

Neg接口 包含适用于数字类型的操作
负数,包括否定、减法和绝对值

Integral接口 包含适用于整数类型的操作

Fractional接口 包含对数字类型起作用的操作
可以分成小数

下面的清单显示了这些接口是如何定义的。



了解哪些操作适用于哪些类型很有用。表 7.1 总结 Prelude 中每个存在的数字接口和实现。

表 7.1 数值接口及其实现总结

界面	描述	实现
在一个	所有数字类型	整数、整数、Nat、双精度
否定	可以为负数的数字类型	整数、整数、双精度
不可缺少的	整数类型	整数、整数、Nat
分数	可分为分数的数值类型	双倍的

一种特别值得注意的方法是 fromInteger。中的所有整数文字 使用 fromInteger 将 Idris 隐式转换为适当的数字类型。作为一个结果，只要有数字类型的 Num 实现，就可以使用该类型的整数文字。

通过对 Num 和相关接口进行新的实现，您可以使用标准您自己的类型的算术符号和整数文字。例如，清单 7.9 显示表示算术表达式的 Expr 数据类型，包括“绝对值”运算和计算表达式计算结果的 eval 函数。

清单 7.9 算术表达式数据类型和求值器 (Expr.idr)

```

数据 Expr num = Val num
      | 添加 (Expr num) (Expr num)
      | 低于 (Expr num) (Expr num)
      | 很多 (Expr num) (Expr num)

```

表达式由数字文字的类型参数化。

```

| Div (Expr num) (Expr num)
| 绝对值 (Expr num)

eval : (Neg num, Integral num) => Expr num -> num
评估 (Val x) = x
评估 (加 xy) = 评估 x + 评估 y
评估 (子 xy) = 评估 x - 评估 y
评估 (Mul xy) = 评估 x * 评估 y
eval (Div xy) = eval x `div` eval y
评估 (Abs x) = 绝对 (评估 x)

```

您需要 num 是可否定的,因为评估器减去,并且还支持整数除法。

要构造表达式,您需要直接应用Expr的构造函数。为了例如,要表示表达式 $6 + 3 * 12$ 并对其求值,您需要编写如下:

```

*Expr> 添加 (Val 6) (Mul (Val 3) (Val 12))
加 (Val 6) (Mul (Val 3) (Val 12)) : Expr 整数

*Expr> eval (加 (Val 6) (Mul (Val 3) (Val 12)))
42 : 整数

```

另一方面,如果您为Expr进行Num实现,您将能够使用标准算术符号(使用+、*和整数文字)来构建类型值经验。此外,如果您进行Neg实现,您将能够使用否定数字和减法。以下清单显示了如何执行此操作。

清单 7.10 Expr (Expr.idn) 的 Num 和 Neg 的实现

```

Num ty -> Num (Expr ty) 其中
  (+) = 添加
  (*) = 我有
  来自整数 = Val 。从整数

Neg ty -> Neg (Expr ty) 其中
  否定 x=0-x
  (-) = 子
  绝对值 = 绝对值

```

您需要约束 Num ty 因为您正在使用 fromInteger 的 ty 实现。

(.) 函数允许您组合函数。

功能组成

(.) 函数为您提供了一个简明的组合两个函数的符号。它有以下类型和定义:

```

(.) : (b -> c) -> (a -> b) -> a -> c
(.) func_bc func_ab x = func_bc (func_ab x)

```

在清单 7.10 的例子中,你可以这样写:

```
fromInteger x = Val (fromInteger x)
```

(.) 函数允许您改为这样编写:

```
fromInteger x = (Val . fromInteger) x
```

(继续)

最后,您可以使用部分应用程序,而不是将 x 作为双方的参数:

```
来自整数 = Val _从整数
```

要构造表达式并对其求值,您可以使用标准符号:

```
*Expr> (Expr_) (6 + 3 * 12)
加 (Val 6) (Mul (Val 3) (Val 12)) : Expr 整数
```

```
*Expr> 评估 (6 + 3 * 12)
42 整数
```

在第一种情况下,您需要使用来明确数字表达式应解释为Expr,而不是默认值,即Integer。在第二种情况下,Idris从eval的类型推断参数必须是Expr。

7.2.3 使用 Cast 进行类型之间的转换

在第2章中,您看到了cast函数,它用于在不同的兼容类型之间转换值。如果您查看cast的类型,您会发现它有一个使用Cast接口的受约束的泛型类型:

```
伊德里斯> :t 演员表
cast : 从 to => 从 -> to 转换
```

与我们目前看到的接口不同, Cast有两个参数而不是一个。下一个清单给出了它的定义。Idris中的接口可以有任意数量的参数(甚至为零!)。

清单 7.11 Cast接口 (在 Prelude 中定义)

```
接口从哪里投到哪里
cast : (orig : from) -> to
```

Cast有两个参数from和to,它们都是Type类型。

正如我们在第2章中所观察到的,使用cast的转换可能是有损的。例如, Idris 定义了Double和Integer之间的强制转换,这可能会丢失精度。cast的目的是为转换提供一个方便的泛型函数,并具有易于记忆的名称。

要定义具有多个参数的接口的实现,您需要提供两个具体参数。例如,您可以定义从Maybe elem到List elem的强制转换,因为您可以将Maybe elem视为两者之一的列表

零或一元素:

```
Cast (Maybe elem) (List elem) where
  cast Nothing = []
  cast (Just x) = [x]
```

由 Type -> Type 参数化的接口

您还可以定义另一个方向的演员表,从List elem到Maybe elem,但是这可能会丢失信息,因为您需要决定哪个元素如果列表有多个元素,则取。

练习



1为第 7.2.2 节中定义的Expr类型实现Show。

提示:为了保持简单并避免担心优先级,假设所有子表达式都可以写在括号中。

您可以在REPL上测试您的答案,如下所示:

```
*ex_7_2> 显示 ((Expr_) (6+3*12))
“ (6 + (3 * 12) ) ” :字符串
```

```
*ex_7_2> 显示 ((Expr_) (6*3+12))
“ ( (6 * 3) + 12) ” :字符串
```

2为Expr类型实现Eq。表达式应该被认为是相等的,如果它们评价是平等的。因此,例如,您应该看到以下内容:

```
*Expr> (Expr_) (2 + 4) ==3+3
真:布尔
```

```
*Expr> (Expr_) (2 + 4) ==3+4
假:布尔值
```

提示:从实现头Eq (Expr ty) where 开始,并添加约束当你发现你需要它们时。

3实施Cast以允许从Expr num转换为任何适当约束的类型num。

提示:您需要评估表达式,它应该告诉您num需要什么被约束。

您可以在REPL上测试您的答案,如下所示:

```
*ex_7_2> let x : Expr Integer = 6 * 3 + 12 in the Integer (cast x)
30:整数
```

7.3 类型参数化的接口 -> 类型

到目前为止,在我们看到的所有接口中,参数都是Types。然而,对参数的类型没有限制。特别是,接口通常具有Type -> Type 的参数。例如,您已经

看到以下具有约束类型的函数:

```
映射:函数 f => (a -> b) -> fa -> fb
纯:应用 f => a -> fa
(>>=) : Monad m => ma -> (a -> mb) -> mb
```

在每种情况下,参数f或m代表参数化类型,例如List或IO。

您已经在List的上下文中看到了map ,在IO的上下文中看到了pure和(>>=)。

在本节中,我们将了解如何在 Prelude 中使用接口对这些和其他操作进行一般定义,以及如何将它们应用于您自己的类型的一些示例。

7.3.1 使用 Functor 跨结构应用函数

在第 2 章中,您看到了map函数,它将函数应用于列表中的每个元素:

```
伊德里斯>地图 (*2)[1,2,3,4]
[2, 4, 6, 8] :列出整数
```

但是,我注意到该映射不限于列表,而是使用Functor接口具有受约束的泛型类型。函子允许您在泛型类型中统一应用函数。前面的示例在整数列表中统一应用了“乘以二”函数。

以下清单显示了Functor接口的定义,其中包含map作为它的唯一方法,以及它的List 实现,如 Prelude 中所定义。

清单 7.12 Functor接口和List的实现 (在 Prelude 中定义)

```
interface Functor (f : Type -> Type) where map : (func : a -> b) -> fa -> fb
```

明确给出 f 的类型,因为它不是 Type

函子列表在哪里

```
映射函数 [] = []
映射函数 (x::xs) = 函数 x::
映射函数 xs
```

到目前为止,我们所看到的接口都是由类型为Type 的变量参数化的。但是Functor的参数本身就是一个参数化类型 (比如List)。

当参数具有除Type 以外的任何类型时,您需要显式地给出参数的类型。

为集合数据结构提供Functor的实现通常很有用。例如,下面的清单显示了如何为二叉树定义一个Functor实现,在出现的每个元素上统一应用一个函数

在一个节点。

清单 7.13 Functor for Tree (Tree.idr)的实现

数据树元素 = 空

| 节点 (树元素)元素 (树元素)

函子树在哪里

```
map func Empty = 空
map func (节点左 e 右)
= Node (map func left) (func e) (map func right)
```

在左子树上统一应用 func

将 func 应用于节点上的元素

在右子树上统一应用 func

Prelude尽可能为所有类型提供具有单个类型参数的Functor实现,包括List、 Maybe和IO。如果您导入 Data.Vect,则有也是Vect n 的Functor实现,如下所示。

清单 7.14 向量的Functor实现（在 Data.Vect 中定义）

函数 (Vect n) 其中
地图功能 $\lambda = \lambda$
映射函数 $(x :: xs) = \text{函数 } x :: \text{映射函数 } xs$

这里的 n 是一个隐式参数,意味着此实现适用于任何长度的向量。

实现标头中的参数Vect n 表示您正在定义一个 Functor实现任意长度的向量。隐式的通常规则参数适用,如第 3 章所述:在函数参数位置以小写字母开头的任何名称都被视为隐式参数。因此, n是在这里被视为一个隐含的论点。

7.3.2 使用 Foldable 减少结构

如果您以 $1 :: 2 :: 3 :: 4 :: []$ 的形式写下数字列表,您可以计算通过应用以下规则计算数字的总和:

- 1 将每个::替换为+ (给出 $1 + 2 + 3 + 4 + []$)。
- 2 将[]替换为0 (给出 $1 + 2 + 3 + 4 + 0$)。
- 3 计算结果表达式的值 (给出10)。

或者,您可以通过应用以下规则来计算数字的乘积:

- 1 将每个::替换为* (给出 $1 * 2 * 3 * 4 * []$)。
- 2 将[]替换为1 (给出 $1 * 2 * 3 * 4 * 1$)。
- 3 计算结果表达式的值 (给出24)。

通常,我们通过替换[]将列表的内容减少为单个值使用默认值或初始值,并将::替换为具有两个参数的函数,它将每个值与减少列表其余部分的结果相结合。Idris 提供了两个高阶函数,称为folds来建议将结构折叠成单个值,以完成此操作:

```
foldr : (elem -> acc -> acc) -> acc -> 列出 elem -> acc
foldl : (acc -> elem -> acc) -> acc -> 列出 elem -> acc
```

foldr和foldl之间的区别在于结果表达式如何括起来。

在我们的第一个示例中, foldr将计算结果为 $1 + (2 + (3 + (4 + 0)))$,而 foldl将计算结果为 $((0 + 1) + 2) + 3) + 4$ 。换句话说, foldr从左到右处理元素,而foldl从右到左处理元素。

FOLDL 和 FOLDR 中的 TYPE VARIABLES 类型中变量的名称
foldl和foldr表明了它们的用途: elem是 list 和acc是结果的类型。名称acc暗示了一个类型累积参数,在其中计算最终结果。

在每种情况下,第一个参数是要应用的函数(或运算符),第二个参数是初始值。因此,您可以按如下方式计算前面的两个示例:

```
Idris> 文件夹 (+) 0 [1,2,3,4]
```

10:整数

```
Idris> 文件夹 (*) 1 [1,2,3,4]
```

24:整数

或者您可以使用折叠来计算列表字符串中字符串的总长度。

例如, [One , Two , Three] 的总长度应该是11。

让我们在名为 Fold.idr 的文件中使用foldr以交互方式编写此代码: 1类型, 定义 - 您可以从

类型和候选定义开始,并将foldr应用于输入列表xs,但为函数和初始值留有孔值,以便他们的类型可以为您提供有关如何进行的提示:

```
totalLen : 列表字符串 -> Nat totalLen xs = foldr ?sumLength ?
initial xs
```

2 Type, refine - ?initial的类型告诉您初始值必须是Nat:

```
xs : 列表字符串 t : 类型 -> 类型
elem : 类型
```

首字母 :Nat

您可以使用0 对其进行初始化,因为空字符串列表的总长度为0:

```
totalLen : 列表字符串 -> Nat totalLen xs = foldr ?sumLength
0 xs
```

3 Type, refine - ?sumLength的类型和上下文告诉您需要pro
视频功能:

```
xs : 列表字符串 t : 类型 -> 类型
elem : 类型
```

sumLength : 字符串 -> Nat -> Nat

您需要提供的函数接受一个String (代表列表中给定位置的字符串)和一个Nat (代表折叠列表其余部分的结果),并返回一个Nat (代表总长度) .您可以按如下方式完成定义:

```
totalLen : 列表字符串 -> Nat totalLen xs = foldr (\str, len =>
length str + len) 0 xs
```

您可以在REPL 中测试生成的函数:

```
*Fold> totalLen [ “一”、 “二”、“三” ]
```

11:纳特

在前面的示例中,我给出了一个特定于List的foldr类型。但是如果你查看REPL 中的类型,你会看到一个使用Foldable接口的受约束的泛型类型:

```
foldr : 可折叠 t => (elem -> acc -> acc) -> acc -> t elem -> acc
```

结构的可折叠实现解释了如何使用初始值和函数将每个元素与整体折叠结构结合起来,将该结构简化为单个值。下面的清单给出了接口定义。

foldl有一个默认定义,用foldr 编写,所以只需要实现foldr。

清单 7.15 Foldable接口 (在 Prelude 中定义)

```
接口可折叠 (t:类型->类型)其中
foldr : (elem -> acc -> acc) -> acc -> t elem -> acc
foldl : (acc -> elem -> acc) -> acc -> t elem -> acc
```



折叠结构,从左到右工作
折叠结构,从右到左工作。实现 foldl 是可选的。

下一个清单显示了如何在List的 Prelude 中实现Foldable。请特别注意foldr和foldl的实现之间的区别。

清单 7.16 Foldable for List 的实现 (在 Prelude 中定义)

```
可折叠列表 where
foldr func acc [] = acc
foldr func acc (x :: xs) = func x (foldr
func acc xs)

foldl func acc [] = acc
foldl func acc (x :: xs) = foldl func (func acc x) xs
```



将函数应用于第一个元素和结果
折叠列表的尾部
递归折叠尾部,通过将函数应用于尾部和第一个元素来更新初始值

因为我们的Tree数据类型是包含值集合的泛型类型,所以您应该能够提供Foldable实现:

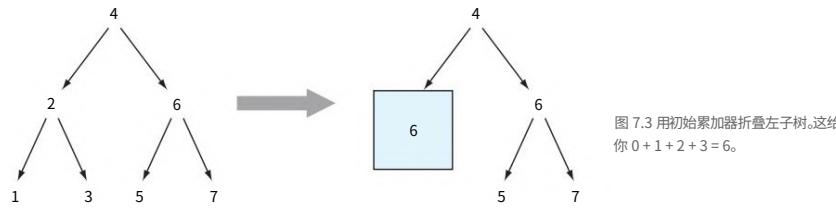
1种类型 首先提供实现标头和骨架定义的文件夹。您可以使用foldl 的默认定义:

```
可折叠树在哪里
foldr func acc 树 = ?Foldable_rhs_1
```

2定义、细化树上的案例拆分。如果树为空,则返回初始值价值:

```
可折叠树在哪里
foldr func acc 空 = acc
foldr func acc (节点左 e 右)
= ?Foldable_rhs_3
```

3 Refine 在Node的情况下,因为foldr从左到右工作,你首先递归地折叠左子树。图 7.3 举例说明了这一点,假设初始累加器为0。

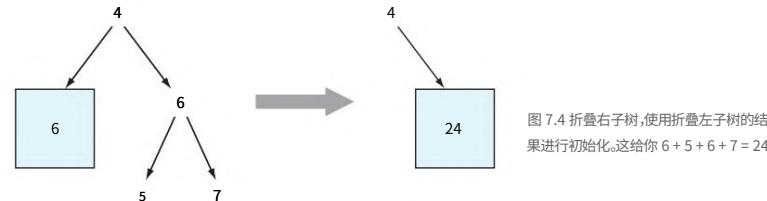


在代码中,它看起来像这样:

可折叠树在哪里

```
foldr func acc 空 = acc
foldr func acc (节点左 e 右)
= let leftfold = foldr func acc 留在
?Foldable_rhs_3
```

4 Refine 接下来,取折叠左子树 (leftfold) 的结果并将其用作折叠右子树时的初始值。图 7.4 说明了这一点同样的例子。



在代码中,它看起来像这样:

可折叠树在哪里

```
foldr func acc 空 = acc
foldr func acc (节点左 e 右)
= let leftfold = foldr func acc 左
rightfold = foldr func leftfold right in
?Foldable_rhs_3
```

5 Refine 最后,将func应用于节点e处的值和折叠的结果右子树:

可折叠树在哪里

```
foldr func acc 空 = acc
foldr func acc (节点左 e 右)
= let leftfold = foldr func acc 左
rightfold = foldr func leftfold right in
功能右折
```

在示例中,这会为您提供结果 $4 + 24 = 28$ 。

7.3.3 使用 Monad 和 Applicative 的通用 do 表示法

还有另外两个有用的接口,您已经了解了

在实践中已经看到: Monad和Applicative。在大多数情况下,您不太可能
需要提供您自己的实现,但它们确实出现在 Prelude 和基础库中,因此了解它们的功能非常有用,尤其是哪些

他们提供的功能。

在第 5 章中,您看到了($>=$)函数是如何用于对IO操作进行排序的:

$(>=) : \text{I a} \rightarrow (\text{a} \rightarrow \text{I b}) \rightarrow \text{I b}$

然后,在第 6 章中,您看到了($>=$)函数如何用于对Maybe计算进行排序,如果任何计算返回Nothing,则放弃排序:

$(>=) : \text{也许 a} \rightarrow (\text{a} \rightarrow \text{也许 b}) \rightarrow \text{也许 b}$

鉴于这两个函数的类型相似 (直接用Maybe代替IO)

和类似目的 (对交互动作或计算进行排序)

可能会失败),您可能希望它们在通用接口中定义。下面的清单显示了提供($>=$)的Monad接口,以及一个join方法

结合了嵌套的单子结构。¹

清单 7.17 Monad 接口 (在 Prelude 中定义)

```
interface Applicative m => Monad (m : Type -> Type) 其中
  ( $>=$ ) : ma -> (a -> mb) -> mb
  加入:m (我的) -> 我的
```

Applicative 接口支持功能
泛型类型中的应用程序。

($>=$)和join都有默认定义,因此您可以根据其中任何一个定义Monad实现。在这里,我们将专注于($>=$)。

在第 6 章中,您看到了 Maybe 的($>=$)定义。在实践中,它定义在前奏如下:

Monad 也许在哪里

$(>=)$ 没有下一个 = 没有
 $(>=)$ (只是 x) 下一个 = 下一个 x

您还看到了纯函数,特别是在IO程序的上下文中,它在IO计算中产生一个值而不描述任何操作:

以及: $\text{a} \rightarrow \text{I a}$

与($>=$)一样, pure也适用于Maybe 的上下文,将Just应用于其参数:

```
Idris> the (Maybe_) (纯 “驱雪” )
只是 “驱雪” :也许是字符串
```

¹ 我们不会在这里详细介绍join ,但它允许您通过连接列表的列表等方式为List定义Monad实现。

同样，考虑到pure在多个上下文中工作，您可能希望它在接口中定义。下面的清单显示了Applicative接口的定义，它提供了一个纯函数和一个($\langle * \rangle$)函数，该函数在结构内应用函数。

你会在第 12 章看到一个Applicative的例子。

清单 7.18 Applicative接口（在 Prelude 中定义）

```
接口 Functor f => Applicative (f : Type -> Type) 其中
  纯:a -> fa (<*>): f (a -> b) -> fa
    -> fb
```

Monad 和 Applicative 有几种实际用途，尽管作为库的用户，您通常只需要知道提供的类型是否有 Monad 和 Applicative 实例，特别是($>>=$)的效果如何运营商是。

Prelude 中一个有趣的 Monad 实现是 List。 List 的($>>=$)函数将输入列表中的每个值依次传递给下一个函数，并将结果组合到一个新列表中。我不会在本书中详细介绍这一点，但您可以使用它来编写非确定性程序。

关于接口的主题还有很多话要说，尤其是我们在本节中简要介绍的层次结构，涵盖 Functor、Foldable、Applicative 和 Monad。深入的讨论超出了本书的范围，但我们在本章中讨论的接口是您最常遇到的接口。

练习

1 为 Expr 实现 Functor（在第 7.2.2 节中定义）。

您可以在 REPL 上测试您的答案，如下所示：

```
*Expr> map (*2) ((Expr_) (1 + 2 * 3))
加 (Val 2) (Mul (Val 4) (Val 6)) : Expr 整数

*Expr> 地图显示 ((Expr_) (1 + 2 * 3))
Add (Val 1 ) (Mul (Val 2 ) (Val 3 )) : Expr String
```

2 为 Vect 实现 Eq 和 可折叠。

提示：如果你导入 Data.Vect，这些实现已经存在，所以你需要手动定义 Vect 来尝试这个练习。

您可以在 REPL 上测试您的答案，如下所示：

```
*ex_7_3> foldr (+) 0 ((Vect _) [1,2,3,4,5])
15 整数

*ex_7_3> (Vect _) [1,2,3,4] == [1,2,3,4]
真：布尔

*ex_7_3> (Vect _) [1,2,3,4] == [5,6,7,8]
假：布尔值
```

7.4 总结

通用类型可以使用接口进行约束。接口描述了可以在特定上下文中应用的函数组。Eq接口提供了比较值是否相等和

不等式。

Ord接口提供了比较两个值以查看哪个更小或更大的功能。接口的实现描述了如何在 spe 中评估接口

具体的上下文。

接口和实现本身可以有约束。约束说明必须实现哪些接口才能使定义有效。Prelude 提供了几个标准接口，包括用于将值转换为字符串的Show，用于算术运算和数字文字的Num，以及用于在类型之间转换的Cast。

接口可以由任何类型的值参数化。前奏中的几个

由Type -> Type 类型的值参数化。

一个结构的Functor实现允许一个统一的动作

应用于结构中的每个元素。

Foldable的实现允许将结构简化为单个
价值。

Monad的一个实现允许您使用do表示法对计算机进行排序
结构上的tations。

平等:表达数据之间的关系

本章涵盖

表达函数和数据的性质检查和保证之间的相等性

数据

用空类型Void显示不可能的情况

您现在已经看到了几种方式,一等类型增加了类型系统的表现力,以及我们赋予函数的类型的精度。您已经了解了如何在类型中使用更高的精度(以及空洞)来帮助编写函数,以及如何编写函数来计算类型。另一种可以使用一等类型来提高类型精度并提高对函数正确运行的信心的方法是编写类型来专门表达数据的属性和数据之间的关系。

在本章中,我们将看一个简单的属性,使用类型来表示值相等的保证。您还将看到如何表示值不相等的保证。有时需要诸如相等和不等之类的属性

您正在使用依赖类型定义更复杂的函数,其中值之间的关系对 Idris 来说可能不会立即显而易见。例如,当我们在向量上定义反向时,您将看到,输入和输出向量长度必须相同,因此我们需要向编译器解释为什么要保留长度。

我们将从查看我们已经使用过的函数exactLength 开始,并详细了解如何从第一原则构建它。

8.1 保证具有相等类型的数据等价

当你想比较值是否相等时,你可以使用`==`运算符,它返回一个Bool 类型的值,给定类型ty中的两个值,其中一个Eq接口的实现:

`(==)`: 等于你 => 你 -> 你 -> 布尔

但是如果你仔细观察这个类型,它告诉你输入 (ty 类型)和输出 (Bool 类型)之间的关系是什么?

事实上,它什么也没告诉你!如果不查看具体的实现,您将无法确切知道`==`的行为方式。对于Eq接口中`==`的实现,以下任何一种行为都是合理的行为,至少就类型而言:

总是返回True 总是返回False

返回输入是否不相等

如果`==`以任何这些方式表现,程序员会感到惊讶,但就 Idris 类型检查器而言,除了在类型中明确给出的假设之外,它不能做出任何假设。因此,如果您想在类型级别比较值,您将需要更具表现力的东西。

事实上,你已经看到了一个你可能想要这样做的例子:在第 5 章的最后,你使用了exactLength函数来检查一个

Vect有一个特定的长度:

精确长度: (len:Nat) -> (输入:Vect ma) -> Maybe (Vect len a)

您使用此函数来检查用户输入的两个向量是否具有完全相同的长度。给定一个特定的长度len和一个长度为 m 的向量,它返回以下内容:

什么都没有,如果len和m不一样只是输入,如

果len和m一样

在本节中,我们将了解如何通过将相等表示为一种类型来实现exactLength ,并且您将更详细地了解为什么`==`是不够的。我们将首先尝试使用`==`来实现它,然后看看我们在哪里遇到了限制。

8.1.1 实现exactLength,第一次尝试

与其导入定义精确长度的Data.Vect,不如先手动定义Vect并给出精确长度的类型和骨架定义。下面的清单显示了我们的起点,在一个名为 ExactLength.idr 的文件中。

清单 8.1 Vect的定义以及exactLength (ExactLength.idr)的类型和骨架定义

```
data Vect:Nat -> Type -> Type where
  nil:Vect Z a
  (::):a -> Vect ka -> Vect (S k) a

exactLength : (len : Nat) -> (input : Vect ma) -> Maybe (Vect len a) exactLength len input = ?exactLength_rhs
```

您应该期望能够通过将输入的长度与所需的长度len进行比较来实现精确长度。如果相等,那么input的长度是len,所以它的类型应该被认为等同于Vect len a,你可以直接返回它。

否则,您将返回Nothing。

作为第一次尝试,您可以尝试以下步骤:

1 定义 因为m没有出现在骨架定义的左侧,所以您需要明确地将其引入范围以便比较len和m。您可以通过将定义更新为以下内容来做到这一点:

精确长度 {m} len 输入 = ?exactLength_rhs

回想一下第 3 章,如果m等隐式参数写在左侧的大括号内,则可以在定义中使用它。

2 定义 您现在可以使用==比较m和len的相等性并检查
产生一个案例陈述:

```
exactLength : (len : Nat) -> (input : Vect ma) -> Maybe (Vect len a) exactLength {m} len input = case m == len of False => ?
  exactLength_rhs_1 True => ?exactLength_rhs_2
```

3 Refine - 对于?exactLength_rhs_1,长度不同,因此您返回
没有什么:

```
exactLength {m} len 输入 = case m == len of False => Nothing True => ?
  exactLength_rhs_2
```

4 Refine - 对于?exactLength_rhs_2,您希望返回Just input ,因为
长度相等。你可以从Just 开始:

```
exactLength : (len : Nat) -> (input : Vect ma) -> Maybe (Vect len a) exactLength {m} len input = case m == len of False =>
  Nothing True => Just ?exactLength_rhs_2
```

5类型 然后,你可以看看范围内变量的类型,以及填充?exactLength_rhs_2洞所需的类型:

```
a : 类型 m : Nat
仅:纳特
输入.:Vect ma
-----
exactLength_rhs_2 : Vect len a
```

即使您已经使用==检查了m和len是否相等,但您不能用输入填充该孔,因为它的类型为Vect m a,并且所需的类型是Vect len a。问题在于,在第5章末尾定义zipInputs时, ==的类型不足以保证m和len相等,即使它返回True。

因此,您将需要考虑替代方法来实施精确长度,在比较m和len时使用信息量更大的类型。

变量的类型告诉Idris变量可以有哪些可能的值,但它没有说明值的来源。如果变量具有Bool类型,Idris知道它可以具有值True或False,但对产生该值的计算一无所知。除了测试相等性之外,还有许多可能的计算可能会产生Bool类型的结果。此外,相等性是由Eq接口定义的,并且类型中没有关于如何实现该接口的保证。

相反,您需要为相等测试创建一个更精确的类型,该类型保证两个输入之间的比较只有在输入确实相同时才能成功。在本节的其余部分,您将了解如何从第一原则开始执行此操作,以及如何使用新的相等类型来实现精确长度。

8.1.2 将 Nats 的相等表示为一种类型

清单8.2显示了一个依赖类型,EqNat。它有两个数字作为参数, num1和num2。如果您有一个EqNat num1 num2类型的值,您知道num1和num2必须是相同的数字,因为唯一的构造函数Same只能构建具有EqNat num num形式的东西,其中两个参数相同。

清单 8.2 将相等的Nats表示为一个类型 (EqNat.idr)

```
数据 EqNat : (num1 : Nat) -> (num2 : Nat) -> 键入 where
    相同 : (num : Nat) -> EqNat num num
```

唯一的构造函数 Same 构造了 num 等于它自己。

对于依赖类型,您可以使用EqNat等类型来表达有关其他数据的附加信息,在这种情况下,表示两个Nat保证相等。

正如您很快就会看到的,这是一个强大的概念,可能需要一些时间才能完全理解。因此,我们将深入研究表示和检查等式。

首先,要了解EqNat的工作原理,让我们在REPL中尝试一些示例:

```
*EqNat> 相同 4
相同 4 : EqNat 4 4

*EqNat> 相同 5
相同 5 : EqNat 5 5

*EqNat> (EqNat 3 3) (相同_)
相同 3 : EqNat 3 3

*EqNat> 相同 (2 + 2)
相同 4 : EqNat 4 4
```

无论您尝试什么,都会重复类型中的参数。然而,编写具有不相等参数的类型是完全有效的:

```
*EqNat> EqNat 3 4
EqNat34:类型
```

但是如果你尝试用这种类型构造一个值,你就不能成功,并且总是会得到一个类型错误:

```
*EqNat> (EqNat 3 4) (Same _)
Prelude.Basics.the 的参数值时:
    类型不匹配
        EqNat num num (相同类型的 num)
    和
        EqNat 3 4 (预期类型)

    具体来说:
        类型不匹配
            0
    和
            1
```

此错误消息表明3和4需要相同,因为它们都需要实例化为Same类型的num。EqNat 3 4类型是一个空类型,这意味着没有该类型的值。

错误消息的特殊性您会注意到,在错误消息中,Idris 通常以两种方式报告错误。第一部分给出了整体类型不匹配。但是,这可能会变得非常大,因此 Idris 还会报告表达式中不匹配的特定部分。在这里,它报告了0和1之间的不匹配,因为Nat是根据Z和S 定义的。总体不匹配在S (S (SZ))和S (S (S (SZ))) 之间,其中具体区别在于Z和S Z之间。

8.1.3 Nats 相等性测试

我们将使用EqNat来帮助实现精确长度。因为EqNat num1 num2本质上是num1必须等于num2的证明,所以我们将编写一个函数来检查输入长度是否相等,如果相等,则将该相等表示为EqNat 的一个实例。

我们将首先编写一个checkEqNat函数,该函数以 EqNat 的形式返回其输入相同的证明,或者如果输入不同则返回Nothing。它有以下类型:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Maybe (EqNat num1 num2)
```

实施后,它将如以下示例所示:

```
*EqNat> checkEqNat 5 5
Just (Same 5) : Maybe (EqNat 5 5)

*EqNat> checkEqNat 1 2
Nothing : Maybe (EqNat 1 2)
```

因为如果num1和num2相同,我们只能为特定的num1和num2拥有EqNat num1 num2类型的值,所以checkEqNat的类型保证如果它成功(即返回Just p形式的值),那么它的输入确实必须相等。

您可以按如下方式实现该功能:

1类型 - 从类型和骨架定义开始:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Maybe (EqNat num1 num2) checkEqNat num1 num2 = ?checkEqNat_rhs
```

2定义 - 您可以通过对两个Nat输入进行大小写拆分来定义函数。首先,在num1 上区分大写:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Maybe (EqNat num1 num2) checkEqNat Z num2 = ?checkEqNat_rhs_1
checkEqNat (S k) num2 = ?checkEqNat_rhs_2
```

3定义然后,在这两种情况下在num2上进行大小写拆分:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Maybe (EqNat num1 num2) checkEqNat ZZ = ?checkEqNat_rhs_3
checkEqNat Z (S k) = ?checkEqNat_rhs_4 checkEqNat (S k) Z = ?checkEqNat_rhs_1 checkEqNat (S k) (S j) = ?
checkEqNat_rhs_5
```

4 Refine 如果您查看?checkEqNat_rhs_3孔的类型,您会发现您需要提供0与自身相同的证据,因此您可以使用Just (Same 0) 填充它。对于?checkEqNat_rhs_4和?checkEqNat_rhs_1,您无法提供任何证据证明0与非零数相同,因此返回Nothing。你现在有这个:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Maybe (EqNat num1 num2) checkEqNat ZZ = Just (Same 0)
checkEqNat Z (S k) = Nothing checkEqNat (S k) Z = Nothing checkEqNat (S k) (S j) = ?checkEqNat_rhs_5
```

5精炼 ?checkEqNat_rhs_5的结果取决于k是否等于j,您可以通过递归调用然后对结果进行大小写拆分来确定:

```
checkEqNat (S k) (S j) = case checkEqNat kj of
    case_val => ?checkEqNat_rhs_5
```

6 定义 case_val上的案例拆分会导致k和不相等 (Nothing)和它们相等 (Just x)的情况。您可以将case split 产生的Just x重命名为更具信息性的名称,并填写递归调用产生Nothing 的 case:

```
checkEqNat (S k) (S j) = case checkEqNat kj of
    没有 => 没有 只是 eq => ?
    checkEqNat_rhs_2
```

7 类型查看checkEqNat_rhs_2的类型可以为您提供一些有关如何继续的信息:

```
k : 晚上
j : Nat eq :
EqNat kj
-----
checkEqNat_rhs_2 : 也许 (EqNat (S k) (S j))

eq的类型是EqNat kj,而您正在寻找Maybe (EqNat (S k) (S j)) 类型的东西。
```

8 Refine 如果k和相等,你知道S k和S j必须相等,所以你可以返回一个用Just 构造的值:

```
checkEqNat (S k) (S j) = case checkEqNat kj of
    没有 => 没有 只是 eq => 只是 ?
    checkEqNat_rhs_2
```

9 细化 - 要完成定义,重命名剩余的孔,并将其提升到顶层定义,您将在下一节中实现。

```
sameS : (eq : EqNat kj) -> EqNat (S k) (S j)

checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Maybe (EqNat num1 num2) checkEqNat ZZ = Just (Same Z) checkEqNat Z (S k) =
Nothing checkEqNat (S k) Z = Nothing checkEqNat (S k) (S j) = case checkEqNat kj of
```

```
    没有=>没有
    Just eq => Just (sameS eq)
```

从提升定义的类型sameS可以看出,它是一个函数,它获取k和j相等的证据,并返回S k和S j相等的证据。通过将数字之间的相等表示为依赖数据类型EqNat,您可以编写类似sameS之类的函数,该函数将EqNat的实例作为输入并对其进行操作,从而从本质上推断出有关相等的附加信息。

8.1.4 作为证明的函数 : 操纵等式

当 k 和 j 不同时 , 不可能创建 EqNat k j 的实例 , 这意味着您可以将 sameS 视为一个证明 , 即如果 k 和 j 相等 , 则 S k 和 S j 也是相等的。

现在让我们尝试实现 sameS 。为了清楚起见 , 我们将使用 Nat 参数明确的 :

```
sameS : (k : Nat) -> (j : Nat) -> (eq : EqNat k j) -> EqNat (S k) (S j)
sameS k j eq = ?sameS_rhs
```

您可以通过以下步骤实现 sameS :

1 定义 - 从创建骨架定义开始 :

```
sameS : (k : Nat) -> (j : Nat) -> (eq : EqNat k j) -> EqNat (S k) (S j)
sameS k j eq = ?sameS_rhs
```

2 定义 接下来 , 在 Atom 中 , 要求对 eq 进行大小写拆分 :

```
sameS : (k : Nat) -> (j : Nat) -> (eq : EqNat k j) -> EqNat (S k) (S j)
相同的 kk (相同的 k) = ?sameS_rhs_1
```

请注意 , k 在该定义的左侧出现了 3 次 ! 因为
你已经用 sameS 类型的 eq 表达了 k 和 j 之间的关系 ,
并且您已经在 eq 上进行了大小写拆分 , Idris 注意到两个 Nat 输入都必须是
相同的。不仅如此 , 如果你尝试给不同的值 , 它会报错。如果 ,
相反 , 你写这个 ,

```
sameS k j (Same k) = ?sameS_rhs_1
```

然后 Idris 将报告以下内容 :

EqNat.idr:15:7: 当检查 sameS 的左侧时 :

类型不匹配
j (推断值)
和
k (给定值)

换句话说 , 因为类型声明两个 Nat 输入必须相同 ,
伊德里斯不高兴他们与众不同。因此 , 恢复到 Idris 在 eq 上的案例拆分后生成的左侧 :

```
sameS : (k : Nat) -> (j : Nat) -> (eq : EqNat k j) -> EqNat (S k) (S j)
相同的 kk (相同的 k) = ?sameS_rhs_1
```

3 Type 如果你检查 ?sameS_rhs_1 的类型 , 你会发现你需要 pro
有证据表明 S k 与自身相同 :

```
k : 晚上
-----
sameS_rhs_1 : EqNat (S k) (S k)
```

4 细化 - 因此您可以按如下方式完成定义 :

```
sameS : (k : Nat) -> (j : Nat) -> (eq : EqNat k j) -> EqNat (S k) (S j)
相同的 kk (相同的 k) = 相同(S k)
```

IDRIS中的证明原则上,您可以陈述并尝试证明 Idris 中任何函数的复杂属性。例如,您可以编写一个函数,其 type 声明将列表反转两次会产生原始列表。然而,在实践中,您很少需要操纵比

相同的实施不过,您将在第 8.2 节中看到更多关于操作等式的内容,以及定义函数时它们出现的位置。

清单 8.3 给出了 checkEqNat 的完整定义,使用的是 sameS 版本
带有明确的论据。您也可以在不使用 sameS 的情况下编写此函数,
而是在 eq 上使用案例拆分。您也可以使用 do 表示法,如最后所述
第 6 章,使定义更加简洁。作为练习,尝试重新实现它
在这些方式中的每一种。

清单 8.3 使用 EqNat (EqNat.idr) 测试 Nats 的相等性

```
sameS : (k : Nat) -> (j : Nat) -> (eq : EqNat k j) -> EqNat (S k) (S j)
相同S kk (相同k) = 相同(S k)

checkEqNat : (num1 : Nat) -> (num2 : Nat) -> 也许 (EqNat num1 num2)
checkEqNat ZZ = Just (Same 0)
checkEqNat Z (S k) = 无
checkEqNat (S k) Z = 无
checkEqNat (S k) (S j) = case checkEqNat k j of
```

如果 k 和 j 相等,则 S k 和 S j 相等。您使用 sameS 来为此提供证据。

没有 => 没有
Just eq => Just (sameS)

Z 和 S 是 Nat 和
永远不可能平等。

-- 当量

与 == 不同, checkEqNat 表示其输入和输出之间的关系
正是在它的类型。使用它,我们可以再次尝试实现精确长度。

8.1.5 实现 exactLength, 第二次尝试

早些时候,在确定布尔比较是不够的之前,我们在实现精确长度时达到了以下几点:

```
精确长度: (len:Nat) -> (输入:Vect ma) -> Maybe (Vect len a)
exactLength {m} len 输入 = case m == len of
    错误 => ?exactLength_rhs_1
    真 => ?exactLength_rhs_2
```

除了使用布尔比较运算符 == 来比较 m 和 len,您还可以
尝试使用 checkEqNat m len。这将返回一个类型为 Maybe (EqNat m len) 的值,所以如果
m 和 len 相等,您将在类型中获得一些附加信息,说明
精确比较结果的含义。在第二次尝试中,您可以
实现功能如下:

1 定义 和以前一样,从类型和骨架定义开始,将 m 带入
左侧范围:

```
精确长度: (len:Nat) -> (输入:Vect ma) -> Maybe (Vect len a)
精确长度 {m} len 输入 = ?exactLength_rhs
```

2 定义 而不是通过对 $m ==$ 的结果进行大小写来定义函数
 len ,对 checkEqNat 的结果进行大小写分割:

```
exactLength : (len : Nat) -> (input : Vect ma) -> Maybe (Vect len a) exactLength {m} len input = case checkEqNat m len of
    nothing => ?exactLength_rhs_1 只是 eq_nat => ?
    exactLength_rhs_2
```

3 Refine - 如果 checkEqNat 返回 Nothing ,则输入不同,因此输入
向量的长度错误:

```
exactLength : (len : Nat) -> (input : Vect ma) -> Maybe (Vect len a) exactLength {m} len input = case checkEqNat m len of
    nothing => 没有 只是 eq_nat => 只是 ?
    exactLength_rhs_2
```

4 类型如果 checkEqNat 返回 Just eq_nat ,则长度相等,并且 eq_nat 提供它们相等的证据。对于?
 $exactLength_{rhs_2}$,你有这个:

```
米:自然
仅:纳特
eq_nat:EqNat m len a:输入类型.Vect
ma
-----
exactLength_rhs_2 : 也许 (Vect len a)
```

5 定义 和以前一样,您仍然需要提供类型为 $\text{Maybe } (\text{Vect } \text{len } a)$ 的结果,而您可用的只是类型为 Vect
 m a 的输入。但是你也有 eq_nat ,它提供了 m 和 len 相等的证据。如果您在 eq_nat 上进行大小写拆
分,您将获得以下信息:

```
exactLength : (len : Nat) -> (input : Vect ma) -> Maybe (Vect len a) exactLength {m} len input = case checkEqNat m len of
    nothing => 什么都没有 (相同的
    长度)>> ?exactLength_rhs_1
```

然后,在检查新孔的类型 $?exactLength_{rhs_1}$ 时,您会看到:

```
米:自然
a:输入类型:
Vect len a
-----
exactLength_rhs_1 : 也许 (Vect len a)
```

因为 eq_nat 只能采用 Same len 的形式,并且 Same len 的类型强制 m 与 len 相同,所以 Idris 将所
需的类型细化为 $\text{Maybe } (\text{Vect } \text{len } a)$ 。

6 细化 从这里,很容易完成定义:

```
exactLength : (len : Nat) -> (input : Vect ma) -> Maybe (Vect len a) exactLength {m} len input = case checkEqNat m len of
    nothing=>没有
    Just (Same len) => 只需输入
```

然而，这并不完全是 Prelude 中使用的定义。相反，Prelude 使用 Idris 内置的通用相等类型。

8.1.6 一般平等`=`类型

Idris 没有为您需要比较的每种可能的类型定义特定的相等类型和函数，例如 Nat 与 EqNat 和 checkEqNat，Idris 提供了一个通用的相等类型。这是 Idris 的语法中内置的，因此您不能自己定义它（因为 = 是保留符号），但从概念上讲，它的定义如下。

清单 8.4 泛型相等类型的概念定义

数据 `(=) : a -> b ->` 键入 where
参考 `x = x`

= 有两个参数:一个是泛型类型 a,另一个是泛型类型 b。
类似 Same,但 x 是隐含参数

Refl 是 reflexive 的缩写，这是一个数学术语，(非正式地) 表示一个值等于它自己。与 EqNat 一样，您可以在 REPL 上尝试一些示例：

伊德里斯> (`Hello = Hello`) Refl
参考：“你好”=“你好”

伊德里斯> (`True = True`) Refl
参考：真=真

如果您将更复杂的表达式作为类型的一部分，Idris 将评估它们。例如，如果您将表达式 $2 + 2$ 作为类型的一部分， $2 + 2 = 4$ ，Idris 将计算它：

伊德里斯> (`2 + 2 = 4`) Refl
参考：4 = 4

和以前一样，如果你尝试使用 Refl 来构建一个使用两个不相等值的相等类型的实例，你会得到一个错误：

Idris> (`True = False`) Refl (input):1:5: 当检查参数值到函数
Prelude.Basics.the: 之间的类型不匹配

`x = x` (参考类型)
和
`True = False` (预期类型)

具体来说：
类型不匹配
真的
和
错误的

使用内置的相等类型，而不是 EqNat，您可以定义 checkEqNat，如下所示。

保证具有相等类型的数据等价

**清单 8.5 使用泛型相等类型检查Nats之间的相等性
(CheckEqMaybe.idr)**

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Maybe (num1 = num2)
checkEqNat ZZ = Just Refl
checkEqNat Z = Nothing
checkEqNat (S k) Z = Nothing
checkEqNat (S k) (S j) = case checkEqNat k j of
```

没有=>没有
只是 prf => 只是 (cong prf)

cong 是 sameS 的
通用版本。

一致性在清单

8.5 中,您使用 cong 将 $k = j$ 类型的值转换为 $S k = S j$ 类型的值。它有以下类型:

cong : {func : a -> b} -> x = y -> func x = func y

换句话说,给定一些 (隐式) 函数 func,如果您有两个相等的值,那么将 func 应用于这些值会给出相等的结果。这与 sameS 执行相同的工作,使用泛型相等类型。

练习



1 实现以下函数,该函数声明如果将相同的值添加到相等列表的前面,则结果列表也相等: same_cons : {xs : List a} -> {ys : List a} ->

xs = ys -> x :: xs = x :: ys

因为这个函数代表一个相等性证明,所以知道你的定义类型检查并且是完全的就足够了:

*ex_8_1> :total same_cons
Main.same_cons 是 Total

2 实现以下函数,该函数说明如果两个值x和y 相等,并且两个列表xs和ys相等,则两个列表x :: xs和y :: ys也必须相等:

same_lists : {xs : 列表 a} -> {ys : 列表 a} ->
x = y -> xs = ys -> x :: xs = y :: ys

同样,知道您的定义类型检查就足够了。

3 定义一个类型ThreeEq,表示三个值必须相等。

提示: ThreeEq的类型应该是 $a \rightarrow b \rightarrow c \rightarrow \text{Type}$ 。

4 实现以下函数,该函数使用ThreeEq:

allSameS : (x, y, z : Nat) -> ThreeEq xyz -> ThreeEq (S x) (S y) (S z)

这种类型是什么意思?

8.2 实践中的相等性:类型和推理相等性证明和操作它们的函数在定义具

有依赖类型的函数时很有用。您在实现checkEqNat时看到了一个小示例,您在其中编写了一个sameS函数来显示将 1 添加到相等的数量会导致相等的数量。

在以数字为索引的类型上编写函数时,通常需要推理相等性。例如,在处理向量时,您可能需要用向量类型中出现的自然数证明两个表达式之间的等价性。要了解这是如何发生的以及如何处理它,我们将看看如何实现一个反转Vect 的函数。

8.2.1 反转向量

如果您导入Data.Vect,您将可以访问一个反转向量的函数,其类型如下:

反向 Vect n elem -> Vect n elem

该类型声明反转向量会保留输入向量的长度。您希望使用以下规则实现此功能相当简单:

如果输入向量为空,则返回一个空向量。 如果输入向量不为空并且
有一个头x和一个尾xs,则反转xs和
附加x。

让我们看看如果我们尝试在文件 ReverseVec.idr 中实现我们自己的版本myReverse会发生什么:

1类型,定义 从类型和骨架定义开始,然后在
输入:

```
myReverse : Vect n elem -> Vect n elem myReverse [] = ?myReverse_rhs_1
myReverse (x :: xs) = ?myReverse_rhs_2
```

2 Refine - 对于?myReverse_rhs_1,返回一个空向量:

```
myReverse : Vect n elem -> Vect n elem myReverse [] = [] myReverse (x :: xs) = ?myReverse_rhs_2
```

3 优化失败 对于?myReverse_rhs_2,您希望能够反转xs和
附加x,如下所示,但不幸的是,这失败了:

```
myReverse : Vect n elem -> Vect n elem myReverse [] = [] myReverse (x :: xs) = myReverse xs ++ [x]
```

此行有类型
错误

错误消息告诉您 Idris 发现您给出的值的类型Vect (k + 1)与预期的类型Vect (S k) 不匹配:

```
ReverseVec.idr:6:12:
使用预期类型检查 myReverse 的右侧时
Vect (S k) 元素

类型不匹配
Vect (k + 1) elem (myReverse xs ++ [x] 的类型)
和
Vect (S k) elem (预期类型)
```

这似乎令人惊讶,因为我们对加法行为的了解表明,无论 k 的值如何, $S k$ 和 $k + 1$ 的计算结果总是相同的。

我们将推迟这个定义的完成,并仔细研究 Idris 中的类型检查是如何工作的,以便了解哪里出了问题以及我们如何纠正它。

8.2.2 类型检查和评估

当 Idris 对表达式进行类型检查时,它会查看表达式的预期类型,并在评估两者后检查给定表达式的类型是否匹配。以下代码片段将进行类型检查:

```
test1 : Vect 4 Int
测试1 = [1, 2, 3, 4]

test2: 权重 (2 + 2) Int
测试2 = 测试1
```

尽管 test1 和 test2 在它们的类型中有不同的表达式,但这些表达式的计算结果相同,因此您可以定义 test2 以返回 test1。

您可以在REPL中看到Idris 类型检查器如何通过使用匿名函数 (在第 2 章中解释)来评估包含类型变量的类型,以引入具有未知值的变量。考虑这个例子:

```
*ReverseVec> \k, elem => Vect (1 + k) elem \k => \elem => Vect (S k) elem : Nat ->
Type -> Type
```

在这里,Idris 将类型中的 $1 + k$ 评估为 $S k$ 。但是,如果您尝试将参数交换为+,您将得到不同的结果:

```
*ReverseVec> \k, elem => Vect (k + 1) elem \k => \elem => Vect (plus k 1) elem : Nat ->
Type -> Type
```

在这里,Idris 计算了 $k + 1$ 到 $plus k 1$,其中 $plus$ 是在Nat上实现加法的 Prelude 函数。要查看差异的原因,您需要查看 $plus$ 的定义。您可以使用REPL命令:printdef 执行此操作,该命令打印作为参数给出的函数的模式匹配定义:

```
*ReverseVec> :printdef plus plus : Nat -> Nat -> Nat
plus 0 right = right plus (S left) right = S (plus left right)
```

因为plus是由第一个参数的模式匹配定义的,所以 Idris 无法计算再加上 k 1。为此,它需要知道k采用什么形式,但在时刻,加号的两个子句都不匹配。

回到我们定义myReverse 的问题,我们来看看当前状态:

```
myReverse : Vect n elem -> Vect n elem
我的反向 [] = []
myReverse (x :: xs) = myReverse xs ++ [x]
```

此行有类型
错误

如果您使用let重写定义来构建结果的每个组件,并且给返回值留个洞,你可以更详细地看到你要解决什么问题解决:

```
myReverse : Vect n elem -> Vect n elem
我的反向 [] = []
myReverse (x :: xs) = let rev_xs = myReverse xs
                      结果 = rev_xs ++ [x] 在
                      ?myReverse_rhs_2
```

检查?myReverse_rhs_2的类型会显示每个组件的类型和所需的结果类型:

```
元素:类型
x:元素
k:晚上
xs:Vect k elem
rev_xs:Vect k elem
结果:Vect (加 k 1)元素
-----
myReverse_rhs_2:Vect (S k) 元素
```

对于预期的结果,您有一个类型为Vect (k + 1) elem 的表达式,但是您需要一个类型为Vect (S k) elem的表达式。

你知道,从之前在REPL的评估中, Vect (1 + k) elem 将评估为您需要的类型,因此您需要能够向 Idris 解释1 + k 等于k + 1;或者,当评估时, S k等于加 k 1。Idris 库提供了一个可以帮助您的函数:

```
plusCommutative : (left : Nat) -> (right : Nat) -> left + right = right + left
```

如果您在REPL中检查plusCommutative的类型,其值为1和k表示left和对,分别,你会看到你需要的完全平等:

```
*ReverseVec> :t \k => plusCommutative 1 k
\k => plusCommutative 1 k : (k : Nat) -> Sk = plus k 1
```

您可以将此表达式的类型视为允许您替换的“重写规则”一个值与另一个值。如果您能找到一种方法来应用此规则来重写预期输入Vect (plus k 1) a,就可以完成定义了。

8.2.3 重写构造:使用相等重写类型

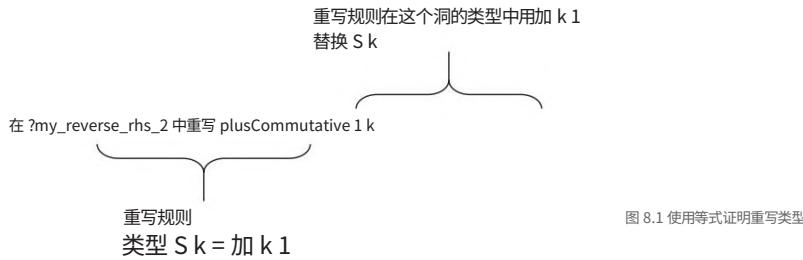
我们将在以下点恢复我们对myReverse的定义,使用let命名我们想要返回的结果:

```
myReverse : Vect n elem -> Vect n elem
myReverse [] = []
myReverse (x :: xs) = let result = myReverse xs ++ [x] in
                      ?myReverse_rhs_2
```

在这个阶段,这些是类型:

```
元素:类型
x:元素
k:晚上
xs:Vect k elem
结果:Vect (加 k 1)元素
-----
myReverse_rhs_2:Vect (S k) 元素
```

为了完成定义,可以使用plus Commutative 1 k给出的信息来重写?myReverse_rhs_2的类型。您需要将?myReverse_rhs_2类型中的任何S k替换为plus k 1,以便您可以返回结果。Idris提供了一种使用等式证明来重写类型的语法,如图 8.1 所示。



因此,您可以使用rewrite构造来实现myReverse以更新?myReverse_rhs_2 的类型,执行以下步骤:

1 定义 在使用plusCommutative 1 k 重写类型之前,您需要将k带入范围,并且k来自输入向量长度的模式匹配:

```
myReverse : Vect n elem -> Vect n elem
myReverse [] = []
myReverse {n = S k} (x :: xs)
```

```
= 让结果 = myReverse xs ++ [x] in
?myReverse_rhs_2
```

2 Refine, type - 现在,您可以应用重写规则:

```
myReverse : Vect n elem -> Vect n elem
myReverse [] = []
myReverse {n = S k} (x :: xs)
```

```
= 让结果 = myReverse xs ++ [x] in
在 ?myReverse_rhs_2 中重写 plusCommutative 1 k
```

如果您查看?myReverse_rhs_2 的结果类型,您可以看到重写的效果:

```
elem : 类型 k : Nat

x : 元素
xs : Vect k elem
结果 : Vect (加 k 1) elem _rewrite_rule:加 k1=Sk

-----
myReverse_rhs_2 : Vect (加 k 1) elem
```

3 Refine 最后,您可以使用表达式搜索来完成定义:

```
myReverse : Vect n elem -> Vect n elem myReverse [] = [] myReverse {n
= S k} (x :: xs)

= 让结果 = myReverse xs ++ [x] in
在结果中重写 plusCommutative 1 k
```

使用重写,您已将类型(S k) 中的表达式替换为等效表达式 (加 k 1),这允许您使用结果。但是,尽管这允许您编写函数,但此定义仍有一些不足之处:计算结果的定义部分(myReverse xs ++ [x])在证明的细节中变得相当丢失。您可以通过使用孔委托证明的细节来改进这一点。

8.2.4 委托证明和重写漏洞

除了在myReverse的定义中应用重写之外,您还可以使用孔和交互式编辑来生成包含证明细节的辅助函数。让我们从myReverse 的初始 (失败)定义重新开始:

```
myReverse : Vect n elem -> Vect n elem myReverse [] = [] myReverse (x :: xs) = myReverse xs ++ [x]
```

您可以使用以下步骤更正此定义:

**1 Refine, type - 在右侧添加一个孔,将初始尝试作为
争论:**

```
myReverse : Vect n elem -> Vect n elem myReverse [] = [] myReverse (x :: xs) = ?reverseProof (myReverse xs ++ [x])
```

如果您检查?reverseProof 的类型,您将确切地看到您需要如何重写类型以使该定义被接受:

```
elem : 类型 x : elem
k : 晚上
```

```

xs : Vect k elem
-----
reverseProof : Vect (plus k 1) elem -> Vect (S k) elem

2键入在 Atom 中使用 Ctrl-Alt-L,将?reverseProof提升到顶级函数:

reverseProof : (x : elem) -> (xs : Vect k elem) ->
              Vect (k + 1) elem -> Vect (S k) elem

myReverse : Vect n elem -> Vect n elem
我的反向 [] = []
myReverse (x :: xs) = reverseProof x xs (myReverse xs ++ [x])

```

3定义定义reverseProof如下,使用rewrite的相同应用

就像您之前对myReverse 的定义一样:

```

reverseProof : (x : elem) -> (xs : Vect k elem) ->
              Vect (k + 1) elem -> Vect (S k) elem
reverseProof {k} x xs 结果 = 在结果中重写 plusCommutative 1 k

```

4 Refine 最后,因为你没有在reverseProof 中使用x或xs ,并且因为

reverseProof将仅由myReverse 使用,您可以将定义整理为
如下:

```

myReverse : Vect n elem -> Vect n elem
我的反向 [] = []
myReverse (x :: xs) = reverseProof (myReverse xs ++ [x])
在哪里
reverseProof : Vect (k + 1) elem -> Vect (S k) elem
reverseProof {k} 结果 = 在结果中重写 plusCommutative 1 k

```

通过引入?reverseProof 漏洞,您可以将myReverse的相关计算部分与证明的细节分开。

8.2.5 附加向量,重温

您通常可以通过注意编写方式来避免重写类型的需要
函数类型。例如,在第 4 章中,你看到了如何定义一个函数
附加具有以下类型的向量:

附加:Vect n elem -> Vect m elem -> Vect (n + m) elem

返回类型中+运算符的参数顺序很重要

因为+ 的定义。如果你开始实现这个 (在文件 AppendVec
.idr) 通过创建骨架定义然后对第一个参数进行大小写拆分,
您将达到以下状态:

```

附加:Vect n elem -> Vect m elem -> Vect (n + m) elem
追加 [] ys = ?append_rhs_1
附加 (x :: xs) ys = ?append_rhs_2

```

然后,如果您检查?append_rhs_1 的类型,您将看到以下内容:

元素:类型
米:自然
ys : 向量

append_rhs_1 : 向量

在这种情况下,第一个参数 [] 的类型为 $\text{Vect } 0 \text{ elem}$,第二个参数 ys ,具有类型 $\text{Vect } m \text{ elem}$ 。根据append的返回类型, $?append_rhs_1$ 应该具有 $\text{Vect } (0 + m) \text{ elem}$ 类型,通过 $+$ 的定义简化为 $\text{Vect } m \text{ elem}$ 。

看看如果交换参数 n 和 m 会发生什么,如下所示:

```
附加 :Vect n elem -> Vect m elem -> Vect (m + n) elem
追加 [] ys = ?append_rhs_1
附加 (x :: xs) ys = ?append_rhs_2
```

您现在看到 $?append_rhs_1$ 的不同类型:

元素:类型
米:自然
ys : 向量

append_rhs_1 : Vect (加 m 0) elem

在myReverse 的定义中, Idris 不能进一步减少 plus m 0 ,因为 plus 由第一个参数的模式匹配定义,并且 m 的形式是未知的。因此,对于这种情况,您不能简单地返回 ys ,您需要重写 $append_rhs_1$ 的类型。 $append_rhs_2$ 会有类似的问题;以下清单显示了append 的定义,用孔代替了必要的重写。

**清单 8.6 在向量上实现追加,其中+的参数在
返回类型 (AppendVec.idr)**

?append_nil 代表在这里返回 ys 是有
效的证明。

```
附加 :Vect n elem -> Vect m elem -> Vect (m + n) elem
追加 [] ys = ?append_nil ys
附加 (x :: xs) ys = ?append_xs (x :: 附加 xs ys)
```

?append_xs 代表返回
x :: 是有效的证明

在此处附加 xs ys。

下一个清单显示了append 的完整定义,其中定义为
 $append_nil$ 和 $append_xs$ 为每种情况重写类型。

**清单 8.7 通过为append_nil添加重写证明来完成对向量的追加
和append_xs (AppendVec.idr)**

```
append_nil : Vect m elem -> Vect (plus m 0) elem
append_nil {m} xs = 在 xs 中重写 plusZeroRightNeutral m

append_xs : Vect (S (m + k)) elem -> Vect (plus m (S k)) elem
append_xs {m} {k} xs = 重写符号
(plusSuccRightSucc mk) 在 xs
```

sym 是一个函数
这反转了重写的方向。

```
附加 :Vect n elem -> Vect m elem -> Vect (m + n) elem
追加 [] ys = append_nil ys
追加 (x :: xs) ys = append_xs (x :: 追加 xs ys)
```

这些重写使用了 Prelude 中的几个定义。其中两个协助重写
使用Nat的 ing 表达式：

```
plusZeroRightNeutral : (left : Nat) -> left + 0 = left
plusSuccRightSucc : (left : Nat) -> (right : Nat) ->
    S (left + right) = left + S right
```

第三个， sym,允许你反向应用重写规则：

符号:左 = 右 -> 右 = 左

本质上， plusZeroRightNeutral和plusSuccRightSucc一起解释了
如果参数以相反的顺序给出，则plus的行为是相同的。

练习



- 1 使用 plusZeroRightNeutral 和 plusSuccRightSucc, 编写自己的版本
加通勤：

```
myPlusCommutes : (n : Nat) -> (m : Nat) -> n + m = m + n
```

提示：在n上按大小来写。在S k 的情况下，您可以通过对myPlusCommutes k m 的递归调用重写，并且可以嵌套重写。

- 2 你之前写的myReverse的实现效率很低，因为它需要
遍历整个向量以在每次迭代时附加一个元素。你可以
使用辅助函数reverse 编写如下更好的定义，它需要一个
累积参数以构建反向列表：

```
myReverse : Vect na -> Vect na
我的反向 xs = 反向 [] xs
其中反向 :Vect na -> Vect (na+m) a
    反向 acc [] = ?reverseProof_nil acc
    反向 acc (x :: xs)
        = ?reverseProof_xs (reverse (x :: acc) xs)
```

通过实现孔?reverseProof_nil和
?reverseProof_xs。

您可以在REPL上测试您的答案，如下所示：

```
*ex_8_2> myReverse [1,2,3,4]
[4, 3, 2, 1] : Vect 4 整数
```

8.3 空类型和可判定性

您可以使用相等类型= 来编写具有声明两个值的类型的函数
保证相等，然后在程序的其他地方使用此保证。
这是有效的，因为构造具有相等类型的值的唯一方法是使用
Refl，并且Refl只会构造一个类型为x = x 的值：

```
伊德里斯> :t Refl
参考 x = x
```

伊德里斯> Refl {x = 94}
参考 94 = 94

但是,如果您想说相反的话,即保证两个值不相等怎么办?当您在一个类型中构造一个值时,您实际上是在证明该类型的元素存在。为了证明两个值 x 和 y 不相等,您需要能够证明 $x = y$ 类型的元素不存在。

在本节中,您将看到如何使用空类型Void来表示某事是不可能的。如果一个函数返回一个Void类型的值,那只能意味着不可能构造其输入的值(或者,从逻辑上讲,它的输入的类型表达了矛盾。)我们将使用Void来表达类型的保证:值不能相等,然后使用它为checkEqNat编写更精确的类型,从而保证

如果它的输入相等,它将产生一个它们相等的证明如果它的输入不相等,它将产生一个它们不相等的证明

首先,让我们看一下Void在最简单的情况下是如何定义和使用的。

8.3.1 Void:没有值的类型

为了表达不可能发生的事情,Prelude 提供了一个没有值的类型, Void。Void的完整定义如下:

数据无效:键入位置

您不能直接写入Void类型的值,因为没有!因此,如果您有一个返回Void类型的函数,那一定是因为它的一个参数也无法构造。

正如您可以使用=来编写函数来表达有关函数行为方式的事实一样,您也可以使用Void来表达有关函数不行为方式的事实。例如,您可以证明 $2 + 2 \neq 5$ 。

让我们在一个名为 Void.idr 的文件中编写一个函数:

1类型 从编写适当的类型开始:

twoPlusTwoNotFive : 2 + 2 = 5 -> 无效

你可以读到“如果 $2 + 2 = 5$,则返回一个空类型的元素”。

2定义添加骨架定义为您提供:

twoPlusTwoNotFive : 2 + 2 = 5 -> 无效

twoPlusTwoNotFive prf = ?twoPlusTwoNotFive_rhs

如果您查看prf 的类型,您会发现它证明了 $4 = 5$:

prf : 4 = 5

twoPlusTwoNotFive_rhs : 无效

3定义 - 您可以尝试通过prf 上的案例拆分来定义函数。伊德里斯制作这个：

```
twoPlusTwoNotFive : 2 + 2 = 5 -> 无效
twoPlusTwoNotFive Refl 不可能
```

这个定义现在完成了。伊德里斯制作了一个案例，并注意到唯一的可能的输入Refl永远不会有效。回想一下第 4 章，不可能的关键字意味着模式子句不能进行类型检查。

空函数和总函数

重要的是检查返回 Void 的函数是一个总函数，如果你真的很想相信它需要一个不可能的输入：

```
*Void> :total twoPlusTwoNotFive
Main.twoPlusTwoNotFive 是总计
```

否则，您可以编写一个声称通过永远循环返回 Void 的函数：

```
循环：无效
循环=循环
```

这个函数不是全部的，所以你不能相信它真的会产生一个元素虚空之。伊德里斯报告如下：

```
*Void> :总循环
由于递归路径，Main.loop 可能不是全部：
主循环
```

类似地，您可以编写一个函数来显示一个数字永远不会等于它的接班人：

```
valueNotSuc : (x : Nat) -> x = S x -> Void
价值不成功 - 参考不可能
```

如果你能够提供一个空类型的值，你就能够产生一个值任何类型的。换句话说，如果你有一个不可能的值已经发生的证据，你可以做任何事情。 Prelude 提供了一个函数void，它表达了这一点：

无效：无效->一个

编写函数只是为了显示可能看起来很奇怪，而且几乎没有实际用途那件事不可能发生。但如果你知道某事不可能发生，你可以使用这些知识来表达对可能发生的事情的限制。换句话说，你可以更准确地表达函数的意图。

8.3.2 可判定性：精确检查属性

以前，当您编写checkEqNat 时，您使用Maybe作为结果：

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> 也许 (num1 = num2)
```

因此,如果checkEqNat返回Just p 形式的值,则可以确定num1和num2相等,并且p表示它们相等的证明。但是你不能说相反:如果checkEqNat返回Nothing,那么num1和num2保证不相等。

例如,以下定义将是完全有效的,尽管不是很用:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> 也许 (num1 = num2) checkEqNat num1 num2 = 无
```

相反,为了使这种类型更精确,您需要一种方式来说明对于任何一对数字num1和num2,您将始终能够提供它们相等的证明(类型为 $\text{num1} = \text{num2}$)或证明它们不相等(类型为 $\text{num1} = \text{num2} \rightarrow \text{Void}$)。也就是说,您想声明检查 $\text{num1} = \text{num2}$ 是否是可判定的属性。

可判定性 如果您总能说出该属性是否适用于特定值,则某些值的属性是可判定的。例如,检查Nat是否相等是可判定的,因为对于任何两个自然数,您总是可以决定它们是否相等。

清单 8.8 显示了Dec泛型类型,它在 Prelude 中定义。与Maybe 一样, Dec有一个带有值的构造函数(Yes)。与Maybe 不同的是,它还有一个构造函数(No),它带有证明其参数类型的值不能存在的证明。

清单 8.8 Dec:精确说明一个属性是可判定的

► 数据 Dec : (prop : Type) -> Type where
 是的 : (prf : prop) -> Dec prop
 否: (相反:道具->无效) ->十二月道具

Dec prop 类型声明您可以决定 prop 是保证持有还是保证不可能。

contra 是属性 prop 不成立的证明,因为它是一个返回值的函数

空类型 Void,给定一个道具。

prf 是属性 prop 成立的证明。

例如,你可以构造一个 $2 + 2 = 4$ 的证明,所以你会使用Yes:

```
*Void> (Dec (2 + 2 = 4)) (Yes Refl)  
是 Refl : Dec (4 = 4)
```

但是不可能构造一个 $2 + 2 = 5$ 的证明,所以你会使用No并提供你的证据, twoPlusTwoNotFive,这是不可能的:

```
*Void> (Dec (2 + 2 = 5)) (No twoPlusTwoNotFive)  
No twoPlusTwoNotFive : Dec (4 = 5)
```

让我们使用Dec而不是Maybe来重写checkEqNat作为结果类型。这样做,我们将得到 Idris 类型检查器验证的两个保证:

如果输入num1和num2相等,则 checkEqNat保证产生Yes prf 形式的结果,其中prf的类型为num1 = num2。如果输入num1和num2不相等,则 checkEqNat保证生成 No contra 形式的结果,其中contra的类型为num1 = num2 -> Void。

这些是保证,因为checkEqNat的类型在输入和输出类型之间提供了直接链接:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Dec (num1 = num2)
```

您可以在名为 CheckEqDec.idr 的新文件中以交互方式编写此函数:

1 定义、改进 当您使用Maybe定义checkEqNat时,您几乎可以遵循与之前相同的步骤。但是不要使用 Just,而是使用Yes,而不是Nothing,使用No。No需要证明输入不相等,因此您可以暂时为这些留出漏洞:

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Dec (num1 = num2) checkEqNat ZZ = 是 Refl checkEqNat Z (S k) = 否 ?zeroNotSuc
checkEqNat (S k) Z = 否 ?sucNotZero checkEqNat (S k) (S j) = case checkEqNat kj of
```

```
    是 prf => 是 (cong prf)
    没有反对 => 没有 ?noRec
```

即使没有证明,您也可以在REPL上对此进行测试,尽管您可能会在结果中看到漏洞:

```
*CheckEqDec> checkEqNat 3 3
是 Refl : Dec (3 = 3)

*CheckEqDec> checkEqNat 3 4 否 ?noRec : Dec (3 = 4)

*CheckEqDec> checkEqNat 3 0 否 ?sucNotZero : Dec
(3 = 0)
```

2 型 如果您查看孔的类型,您将看到需要证明的内容

完成定义,如下例所示:

```
k : 晚上
-----
zeroNotSuc : (0 = S k) -> Void
```

3 定义、细化 您可以使用 Ctrl-Alt-L 将?zeroNotSuc和?sucNotZero 提升到顶级定义,并使用大小写拆分和不可能来实现它们,就像上一节中的twoPlusTwoNotFive一样,因为零不可能是等于一个非零数:

```
zeroNotSuc : (0 = S k) -> Void zeroNotSuc Refl 不可能
```

```
sucNotZero : (S k = 0) -> Void sucNotZero Refl 不可能
```

4类型,定义对于?noRec,您可以查看类型以了解您需要做什么
节目:

```
k : 晚上
j: 纳特
相反: (k = j) -> 无效
-----
noRec : (S k = S j) -> 无效
```

contra的类型告诉您k保证不等于j。给定

也就是说,你必须证明S k不能等于S j。同样,您可以使用 Ctrl-Alt-L 将其提升到顶级定义:

```
noRec : (对 : (k = j) -> Void) -> (S k = S j) -> Void
noRec 对 prf = ?noRec_rhs_1
```

5定义 noRec的类型表示如果你有一个证明k = j是不可能的,
和S k = S j的证明,你可以产生一个空类型的元素。从逻辑上讲,这两个输入相互矛盾。您可以按情
况编写此函数
在prf上拆分:

```
noRec : (对 : (k = j) -> Void) -> (S k = S j) -> Void
noRec 对 Refl = ?noRec_rhs_1
```

6类型 prf可以取的唯一可能值是Refl,它可以取的唯一方式
如果S k和S j相等,则Refl的值,因此k和相等。认识到这一点后,Idris 改进了contra 的类型,正如您
通过检查
noRec_rhs_1的类型:

```
j: 纳特
对立: (j = j) -> 无效
-----
noRec_rhs_1 : 无效
```

7Refine 要完成定义,可以使用contra生成元素
空类型;表达式搜索会发现:

```
noRec : (对 : (k = j) -> Void) -> (S k = S j) -> Void
noRec 对 Refl = 对 Refl
```

以下清单显示了checkEqNat 的完整定义,包括
辅助函数zeroNotSuc、 sucNotZero和noRec。

清单 8.9 检查Nats是否相等,使用精确类型 (CheckEqDec.idr)

<pre>zeroNotSuc : (0 = S k) -> Void zeroNotSuc Refl 不可能</pre>	<p>给定一个零等于非零数的证明,产生一个空类型的值。</p>
<pre>sucNotZero : (S k = 0) -> Void sucNotZero Refl 不可能</pre>	<p>给定一个非零数等于零的证明,产生一个空类型的值。</p>
<pre>noRec : (对 : (k = j) -> Void) -> (S k = S j) -> Void noRec 对 Refl = 对 Refl</pre>	<p>给定两个数字不相等的证明, 并证明他们的继任者是平等的, 产生一个空类型的值。</p>

空类型和可判定性

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Dec (num1 = num2) checkEqNat ZZ = 是 Refl checkEqNat Z
(S k) = 没有 zeroNotSuc checkEqNat (S k) Z = 没有 sucNotZero checkEqNat (S k) (S j) = case checkEqNat
kj of
```

是 prf => 是 (cong prf)
没有反对 => 没有 (noRec 反对)

用 VOID 证明输入是不可能的当你运行checkEqNat时,你实际上并不会使用zeroNotSuc、sucNotZero或noRec生成一个空类型的值。从本质上讲,产生Void类型值的函数可以被视为不能同时提供所有参数的证明。在noRec的情况下,函数的类型表明,如果你可以同时提供k不等于j的证明和S k = S j的证明,那么就存在矛盾,因此你可以有一个值键入无效。

通过使用Dec,您可以在类型中明确编写checkEqNat应该做的事情:返回输入相等的证明或不相等的证明。

然而,这样做的真正好处不在于checkEqNat本身,而在于使用它的函数,因为它们不仅具有相等测试的结果,而且还证明相等测试已正确运行。

能够保证两个值相等(或不同)在类型驱动开发中通常很有用,因为显示较大结构之间的关系取决于显示各个组件之间的关系。因此,Idris Prelude 提供了一个接口DecEq,它带有一个通用函数decEq,用于确定相等性。

8.3.3 DecEq:可判定相等的接口

Idris Prelude 没有为每种类型提供像checkEqNat这样的特定函数,而是提供了一个接口DecEq。下一个清单显示了DecEq是如何定义的。Prelude 中定义的所有类型都有实现。

清单 8.10 DecEq接口 (在 Prelude 中定义)

给定两个值 val1 和 val2,返回它们相等的证明或它们不同的证明。
 接口 DecEq ty 其中 decEq : (val1 : ty) ->

$$(val2 : ty) \rightarrow \text{Dec} (\text{val1} = \text{val2})$$

您可以使用decEq,而不是定义和使用专用函数checkEqNat来定义确切的长度。以下清单显示了如何执行此操作,它是Data.Vect中使用的精确长度的定义。

清单 8.11 使用decEq (ExactLengthDec.idr) 实现精确长度

```
exactLength : (len : Nat) -> (input : Vect m a) -> Maybe (Vect len a) exactLength {m} len input = case decEq m len of
```

是 Refl => 只需输入
没有反对 => 没有

使用`DecEq`而不是布尔相等运算符`==`,可以为您提供有关相等测试如何工作的强有力的保证。您可以确定,如果`DecEq`返回值

Yes prf的形式,那么输入在结构上确实是相等的。

在下一章中,您将看到如何描述大数据之间的关系
结构(例如显示一个值是列表的一个元素)以及如何使用`DecEq`
建立这些关系的证明。

练习

1 实现以下功能:

```
headUnequal : DecEq a => {xs : Vect na} -> {ys : Vect na} ->
  (相反: (x = y) -> Void) -> ((x :: xs) = (y :: ys)) -> Void
tailUnequal : DecEq a => {xs : Vect na} -> {ys : Vect na} ->
  (相反: (xs = ys) -> Void) -> ((x :: xs) = (y :: ys)) -> Void
```

第一个声明如果两个向量的第一个元素不相等,则向量
必须不相等。第二个状态是如果两个尾部存在差异
向量,则向量必须不相等。

如果您有正确的解决方案,`headUnequal`和`tailUnequal`都应该输入
检查并完全:

```
*ex_8_3> :总头不相等
Main.headUnequal 是 Total
```

```
*ex_8_3> :总尾不相等
Main.tailUnequal 是 Total
```

2 为`Vect`实施`DecEq`。从以下实现标头开始:

```
DecEq a => DecEq (Vect na) 其中
```

提示:你会发现`headUnequal`和`tailUnequal`在这里很有用。记得检查
编写定义时的孔类型。您还应该使用自己的`Vect`定义而不是导入`Data.Vect`,因为该库提供了一个

`DecEq`实现。

您可以在REPL上测试您的答案,如下所示:

```
*ex_8_3> decEq ((Vect _ _) [1,2,3]) [1,2,3]
是 Refl : Dec ([1, 2, 3] = [1, 2, 3])
```

8.4 总结

你可以编写一个类型来表达两个值必须相等的证明。

您可以编写将等式类型作为输入的函数来证明额外的
数据的属性。

使用`Maybe`,您可以在运行时测试值的相等性。

`generic =`类型允许您描述任何类型的值之间的相等性。

当使用依赖类型进行编程时,证明需求自然会出现,
例如显示在反转向量时保留长度。

重写构造允许您使用等式证明来更新类型。 使用漏洞和交互式编辑,可以委托重写的细节

类型到一个单独的函数。

`void`是一个没有值的类型,用来表明一个函数的输入不能全部一下子发生。

如果您总能说出该属性是否适用,则该属性是可判定的
一些特定的值。

使用Dec,你可以在运行时计算一个属性是否保证
持有或保证不持有。

谓词:以类型表达假设和契约

本章涵盖

- 描述和检查向量的成员资格
- 使用谓词使用谓词来
- 描述函数输入和输出的契约
- 推理类型中的系统状态

依赖类型,如EqNat,完全用于描述数 $=$,你在上一章看到的,被使用据之间的关系。这些类型通常称为谓词,它们是完全存在以描述某些数据的属性的数据类型。如果您可以为谓词构造一个值,那么您就知道该谓词所描述的属性必须为真。

在本章中,您将看到如何使用谓词来表达数据之间更复杂的关系。通过在类型中表达数据之间的关系,您可以明确说明您对函数输入所做的假设,并在这些函数出现时让类型检查器检查这些假设

叫。您甚至可以将这些假设视为表达其他参数必须满足的编译时契约,然后才能调用该函数。

在实践中,您经常会编写对一些其他数据的形式做出假设的函数,并事先(希望!)检查这些假设。这里有一些例子:

从文件中读取的函数假定文件句柄描述了一个打开文件。

处理从网络接收到的数据的功能假定数据遵循适当的协议。检索用户数据的功能(例如,对于网站上的客户)假设

用户已成功通过身份验证。

您需要确保在调用这些函数之前检查任何必要的假设,尤其是在安全或用户隐私受到威胁时。在Idris中,您可以在类型中明确表示这种假设,并让类型检查器确保您事先已正确检查了该假设。用类型表达假设的一个优点是它们可以保证成立,即使系统随着时间的推移而发展。

在本章中,我们将深入研究一个特定的小示例:如果该值包含在向量中,则从向量中删除该值。我们将看看如何

表示向量包含特定元素的类型,如何在运行时测试此属性,以及如何在实践中使用这些属性来编写更大的程序,其行为的某些方面以其类型表示。

9.1 隶属度测试 :Elem 谓词

使用=、Void和Dec,您可以描述程序在其类型中满足的属性,从而使类型精确。通常,您将使用=、Void和Dec作为构建块来描述和检查数据片段之间的关系。

描述类型中的关系允许您陈述函数所做的假设,更重要的是,它允许Idris检查是否满足或违反了这些假设。

例如,您可能希望仅在元素存在于

向量中的假设下从向量中删除元素仅在列表已排序的假设下在列表中搜索值仅在以下条件下运行数据库查询假设输入已经

已验证

仅在假设以下情况下向网络上的机器发送消息
你有一个开放的连接

在本节中,我们将更详细地研究其中的第一个。我们将首先尝试编写一个从向量中删除元素的函数,然后我们将了解如何改进类型以声明并检查任何使类型更精确的必要假设。最后,我们将在一个简单的交互式程序中使用该函数。

9.1.1 从 Vect 中移除一个元素

如果您有一个包含许多元素的Vect ,您会期望以下removeElem函数从该向量中删除特定元素：

输入向量应该至少有一个元素。输出向量的长度应该比输入向量的长度小一。

因为Vect由向量中包含的元素类型参数化,并且由向量的长度索引,所以您可以(实际上,您必须)在removeElem类型中表达这些长度属性。您的第一次尝试可能如下所示：

```
removeElem : (value : a) -> (xs : Vect (S n) a) -> Vect na
```

让我们看看如果你尝试编写这个函数会发生什么：

1定义 如果你在xs上添加一个骨架定义和case-split , Idris会给你只有一种模式,因为xs不能是空向量：

```
removeElem : (value : a) -> (xs : Vect (S n) a) -> Vect na
removeElem 值 (x :: xs) = ?removeElem_rhs_1
```

2优化如果您要查找的值等于x,您可以将其从列表中删除,返回xs。在比较value和x之前,您需要优化类型;您可以使用可判定的相等性,以便您可以确定相等性检验是准确的。这是精炼的类型：

```
removeElem : DecEq a => (value : a) -> (xs : Vect (S n) a) -> Vect na
```

3优化 您现在可以使用decEq比较value和x,如果它们是则丢弃x平等的：

```
removeElem : DecEq a => (value : a) -> (xs : Vect (S n) a) -> Vect na
removeElem value (x :: xs) = case decEq value x of Yes prf => xs No
contra => ?removeElem_rhs_3
```

4优化失败 对于?removeElem_rhs_3,您可能希望能够删除递归地从xs获取值,但如果你尝试这个,你会得到一个错误：

```
removeElem : DecEq a => (value : a) -> (xs : Vect (S n) a) -> Vect na
removeElem value (x :: xs) = case decEq value x of
```

```
    是的 prf => xs
    没有 contra => x :: removeElem 值 xs
```

伊德里斯报告如下：

在 removeElem 中检查 Main.case 块的右侧时 removeelem.idr:4:35 与预期类型

Vect na

检查 Main.removeElem 的参数 xs 时：

 类型不匹配
 Vect na (xs 类型)
 和
 Vect (S k) a (预期类型)

问题是removeElem需要一个保证不为空的向量,但xs可能为空!你可以从它的类型Vect n a中看出这一点: n可以代表任何自然数,包括零。

出现这个问题是因为无法保证值会出现在向量中,因此有可能到达向量的末尾而不遇到它。如果发生这种情况,则没有可删除的值,因此您无法满足类型。

您需要进一步细化类型才能编写此函数。您可以尝试以下方法之一:

重写类型,使removeElem返回Maybe (Vect na),如果值没有出现在输入中则返回Nothing。重写类型以便removeElem返回一个依赖对, (newLength **

Vect newLength a)。

在removeElem上表达一个前提条件,即保证该值在向量。

你已经看到了如何用Maybe表示可能的失败(在第4章中),以及如何使用依赖对来表示未知长度(在第5章中)。然而,第三个选项将准确地表达removeElem的目的。为此,您需要编写一个类型来描述值和包含该值的向量之间的关系。

如果您可以在一个类型中描述一个向量包含特定元素,您将能够使用该类型来表达removeElem上的合同,表示您只有在知道该元素在向量中时才能使用它。在本节的其余部分中,您将实现Elem这个类型,使用它来使removeElem的类型更加精确,并了解更多关于如何在实践中使用Elem。

9.1.2 Elem 类型:保证一个值在一个向量中

在上一章中,您看到了如何通过使用特定的EqNat类型或泛型=类型来表示两个值保证相等。其中一种类型的值的存在本质上是两个值相等的证明。你可以做一些类似的事情来保证一个值在一个向量中。

我们的目标是使用以下类型构造函数定义一个类型Elem:

Elem : (value : a) -> (xs : Vect ka) -> 类型

如果我们有一个值和一个包含该值的向量xs,我们应该能够构造一个Elem值xs类型的元素。例如,我们应该能够构建以下内容:

```
oneInVector : Elem 1 [1,2,3] maryInVector :
Elem Mary [ Peter , Paul , Mary ]
```

我们还应该能够构造以下类型的函数,这些函数表明向量中不包含特定值:

```
fourNotInVector : Elem 4 [1,2,3] -> Void
peteNotInVector : Elem  Pete [ John , Paul , George , Ringo ] -> Void
```

下面的清单显示了如何在Data.Vect中定义Elem依赖类型。

**清单 9.1 Elem 依赖类型,表示一个值保证是
包含在向量中 (在 Data.Vect 中定义)**

这里声明 x 是 $x :: xs$ 形式的向量中的第一个值。

```
data Elem : a -> Vect ka -> 输入 where
    这里 : Elem x (x :: xs)
    那里 : (后来 : Elem x xs) -> Elem x (y :: xs)
```

那里指出,如果您知道 x 出现在向量 xs 中,那么 x 也必须出现在向量 $y :: xs$ 中。

这里的值可以被解读为一个值是向量中第一个值的证明,如下所示
例子:

```
oneInVector : 元素 1 [1,2,3]
oneInVector = 这里
```

那里的构造函数,给定一个显示值 x 在向量 xs 中的参数,可以被解读为 x 也必须在向量 $y :: xs$ 中对于任何值 y 的证明。

为了说明这一点,让我们尝试编写maryInVector:

```
maryInVector : Elem Mary [ Peter , Paul , Mary ]
```

1 定义 - 创建右侧带有孔的骨架定义:

```
maryInVector : Elem Mary [ Peter , Paul , Mary ]
maryInVector = ?maryInVector_rhs
```

2 Refine, type 你不能使用Here,因为“Mary”不是向量,所以尝试使用There并为其参数留一个洞:

```
maryInVector : Elem Mary [ Peter , Paul , Mary ]
maryInVector = 有 ?maryInVector_rhs
```

如果你检查?maryInVector_rhs 的类型,你会看到你现在有一个小问题:

```
-----  
maryInVector_rhs : Elem Mary [ Paul , Mary ]
```

3 精炼,输入 如果你再次做同样的事情,应用There并为它留下一个洞参数,你得到以下程序:

```
maryInVector : Elem Mary [ Peter , Paul , Mary ]
maryInVector = 那里 (那里 ?maryInVector_rhs)
```

您现在还有一个更简单的?maryInVector_rhs 类型:

```
-----  
maryInVector_rhs : Elem Mary [ Mary ]
```

成员资格测试 :Elem 谓词

4 Refine “Mary”现在是向量中的第一个元素,因此您可以优化?mary
InVector_rhs使用这里:

```
maryInVector: Elem Mary [ Peter , Paul , Mary ]
maryInVector = 那里 (这里)
```

表达式搜索 使用 Ctrl-Alt-S 的表达式搜索将成功
找到maryInVector 的定义,它通常对构造很有用
Elem和=等依赖类型的值。

您可以使用Elem和类似的类型来描述数据之间的关系来表达
预期作为函数输入的数据形式的合同。这些合同,被
表示为类型,可以由 Idris 使用类型检查来验证;如果函数调用违反合同,程序将无法编译。

更有建设性的是,如果你足够精确地表达你的函数类型,
使用像Elem 这样的类型表示的契约,你知道一旦你的程序编译,每个契约都必须得到满足。您可以使用
Elem来表达合同
removeElem指定输入何时有效。

9.1.3 从 Vect 中移除元素 :类型为契约

我们在编写removeElem时遇到的困难是我们无法进行递归
调用向量的尾部,因为我们不能保证我们是
试图删除是在向量中。因此,我们将改进removeElem 的类型,添加一个参数,该参数表示要删除的元素
必须是
在向量中。以下清单显示了我们的起点。

清单 9.2 从向量中删除一个元素,并在类型中指定一个契约
使用Elem (RemoveElem.idr)

```
removeElem: (值:a) ->
  (xs : Vect (S n) a) ->
  (prf : Elem 值 xs) ->
  Vect na
removeElem 值 xs prf = ?removeElem_rhs
```

在类型驱动开发中,我们的目标是使用更精确的类型来帮助指导
功能的实现。在这里,输入prf为输入提供了更高的精度
removeElem类型,您甚至可以对其进行大小写拆分,以查看您对
从prf指定的关系中输入value和xs。
您可以编写如下函数:

1 定义 - 从prf上的案例拆分开始:

```
removeElem : (value : a) -> (xs : Vect (S n) a) ->
  (prf : Elem 值 xs) ->
  Vect na
removeElem 值 (value :: ys) 这里 = ?removeElem_rhs_1
removeElem value (y :: ys) (后面有) = ?removeElem_rhs_2
```

2 Refine 对于第一种情况, `?removeElem_rhs_1`,您可以从 Idris 生成的模式中看到,如果证明的形式为`Here`,则该值必须是向量中的第一个元素。因此,您可以改进这种情况以删除价值:

```
removeElem : (value : a) -> (xs : Vect (S n) a) ->
            (prf : Elem value xs) ->
            Vect na
removeElem value (value :: ys) Here = ys removeElem value (y :: ys) (there
later) = ?removeElem_rhs_2
```

3类型 如果您查看`?removeElem_rhs_2`的类型,您会发现您似乎遇到了与以前相同的问题,因为`ys`的长度为`n`,并且`removeElem`需要一个长度为`S n`的向量来进行递归调用:

```
a:类型值:a
y:an : nat
ys : Vect na 后来 : Elem
值 ys
-----
removeElem_rhs_2 : Vect na
```

但是您有一些之前没有的进一步信息:您稍后从变量中知道值必须出现在`ys`中,这意味着`ys`必须具有非零长度。但是你怎么能使用这些知识呢?

4定义 鉴于长度`n`必须不为零,诀窍是对`n`进行大小写拆分(通过使用定义左侧的大括号将其带入范围)并表明它不可能为零:

```
removeElem : (value : a) -> (xs : Vect (S n) a) -> (prf : Elem value xs) ->
            Vect na
removeElem value (value :: ys) Here = ys removeElem {n = Z} value (y :: ys)
(后面有) = ?removeElem_rhs_1 removeElem {n = (S k)} value (y :: ys) (后面有) = ?removeElem_rhs_3
```

5优化 现在,您可以完成`?removeElem_rhs_3`。您现在知道`ys`的长度不为零,因此您可以进行递归调用:

```
removeElem : (value : a) -> (xs : Vect (S n) a) -> (prf : Elem value xs) ->
            Vect na
removeElem value (value :: ys) Here = ys removeElem {n = Z} value (y :: ys)
(后面有) = ?removeElem_rhs_1 removeElem {n = (S k)} value (y :: ys) (后面有)
= y :: removeElem value ys 后
```

请注意,您需要稍后传递给递归调用,作为值包含在`ys`中的证据。

6类型,定义 - 对于剩余的孔`?removeElem_rhs_1`,查看其类型以及可用的变量:

```
a:类型值:a

y:a ys :
Vect 0 a later : Elem
value ys
-----
removeElem_rhs_2 : Vect 0 a
```

您正在寻找一个空向量。如果您查看左侧的变量，该空向量应该是从ys中删除值得到的向量。这没有意义，因为ys是一个空向量！

您可以通过在ys上区分大小写来更清楚地说明这一点。伊德里斯只生产一个案子：

```
removeElem : (value : a) -> (xs : Vect (S n) a) -> (prf : Elem value xs) ->
Vect na
removeElem value (value :: ys) 这里 = ys removeElem {n = Z} value (y :: []
) (后面有) = ?removeElem_rhs_1 removeElem {n = (S k)} value (y :: ys) (那里之后)
= y :: removeElem value ys 后
```

查看新孔的类型，?removeElem_rhs_1，显示如下：

```
a:类型值:a

y:a 后来:
元素值 []
-----
removeElem_rhs_1 : Vect 0 a
```

以前，您使用过不可能的关键字来排除不进行类型检查的情况。这种情况下会进行类型检查，因此您不能使用不可能。但是你永远不会有Elem value []类型的值作为输入，因为没有办法构造这种类型的元素。

7精炼 您可以使用前奏曲中定义的荒谬函数来完成定义。它有以下类型：

荒谬:无人居住的 t => t -> a

边栏中描述的Uninhabited接口可以为任何没有值的类型实现（正如您在第8章中看到的twoplus two_not_five）。

所以，你可以细化?remove_Elem_rhs_1如下：

```
removeElem : (value : a) -> (xs : Vect (S n) a) -> (prf : Elem value xs) ->
Vect na
removeElem value (value :: ys) 这里 = ys removeElem {n = Z} value (y :: []
) (后面有) = 荒谬后面 removeElem {n = (S k)} value (y :: ys) (那里later) = y :: removeElem value ys later
```

无人居住界面

如果一个类型没有值,例如 $2 + 2 = 5$ 或 $\text{Elem } x []$,您可以为其提供 `Uninhabited` 接口的实现。前奏曲中对无人居住的定义如下:

```
界面无人居住的地方
无人居住 t -> Void
```

有一种方法,它返回一个空类型的元素。例如,你可以为 $2 + 2 = 5$ 提供 `Uninhabited` 的实现:

```
无人居住 (2 + 2 = 5) 其中
无人居住的 Refl 不可能
```

使用 `uninhabited`,`Prelude` 使用 `void` 将荒谬定义如下:

```
荒谬:无人居住的 t => (h : t) -> a
荒谬的 h = void (无人居住的 h)
```

`removeElem` 的附加参数,类型为 `Elem` 值 `xs`,表示 `removeElem` 可以在 `value` 位于向量 `xs` 中的假设下工作。它也是意味着您必须在调用函数时提供值在 `xs` 中的证明。

9.1.4 自动隐式参数:自动构造证明

如果您尝试使用元素和向量的某些特定值运行 `removeElem`,您会发现您需要为证明提供额外的参数。有时这可能是一个可以构建的证明:

```
*RemoveElem> removeElem 2 [1,2,3,4,5]
removeElem 2 [1, 2, 3, 4, 5] : Elem 2 [1, 2, 3, 4, 5] -> Vect 4 整数
```

有时它可能不会:

```
*RemoveElem> removeElem 7 [1,2,3,4,5]
removeElem 7 [1, 2, 3, 4, 5] : Elem 7 [1, 2, 3, 4, 5] -> Vect 4 整数
```

在第一种情况下,您可以通过显式提供 `Elem 2` 的证明来调用 `removeElem` [1,2,3,4,5]:

```
*RemoveElem> removeElem 2 [1,2,3,4,5] (这里)
[1, 3, 4, 5] : Vect 4 整数
```

像这样明确地提供证明的需要会给程序增加很多噪音,并且会损害可读性。`Idris` 提供了一种特殊的隐式参数,标记为关键字 `auto`,以减少这种噪音。

你可以定义一个 `removeElem_auto` 函数:

```
removeElem_auto : (value : a) -> (xs : Vect (S n) a) ->
    {auto prf : Elem value xs} -> Vect na
removeElem_auto 值 xs {prf} = removeElem 值 xs prf
```

第三个参数名为prf,是一个自动隐式参数。就像您已经看到的隐式参数一样,自动隐式参数可以通过将其写在大括号中来引入范围,并且 Idris 将尝试自动找到一个值。与普通的隐式不同,Idris 将使用与 Atom 中的 Ctrl-Alt-S 进行表达式搜索相同的机器来搜索自动隐式的值。

当您使用value和xs的参数运行removeElem_auto时, Idris 将尝试为标记为auto 的参数构造证明:

```
*RemoveElem> removeElem_auto 2 [1,2,3,4,5]
[1, 3, 4, 5] : Vect 4 整数
```

如果找不到证明,就会报错:

```
*RemoveElem> removeElem_auto 7 [1,2,3,4,5] (input):1:17:当检查参数 prf
到函数 Main.removeElem_auto 时:
找不到类型的值
元素 7 [1, 2, 3, 4, 5]
```

或者,以下清单显示了如何直接使用自动隐式定义removeElem。

清单 9.3 使用自动隐式参数 定义removeElem (RemoveElem.idr)

```
removeElem : (value : a) -> (xs : Vect (S n) a) ->
    {auto prf : Elem 值 xs} ->
    Vect na
removeElem value (value :: ys) {prf = Here} = ys removeElem {n = Z} value (y :: []) {prf =
    There later} = 荒谬后来 removeElem {n = (S k)} value (y :: ys) {prf = 稍后有}
    = y :: removeElem 值 ys
```

prf 是一个自动隐式参数。Idris 将尝试使用表达式
搜索为每个调用自动查找一个值。

Idris 将在此调用中自动找到证明的值
(稍后)。

不过,一般来说,您不会知道要传递给removeElem 的具体值。
它们可以是从用户读取的值或由程序的另一部分构造的值。
因此,我们需要考虑在运行时之前输入未知的情况下如何使用removeElem。

就像您编写了一个checkEqNat函数来确定两个数字是否相等,然后使用接口将其推广到decEq 一样,您将需要一个函数来确定一个值是否包含在向量中。

9.1.5 可判定谓词:判定向量的成员资格

正如您在上一章中所看到的,如果您始终可以判断该属性是否适用于某些特定值,则该属性是可判定的。使用下面的函数,可以看到Elem value xs是针对value和xs的特定值的可判定属性,所以Elem是可判定谓词:

```
isElem : DecEq ty => (value : ty) -> (xs : Vect n ty) -> Dec (Elem value xs)
```

`isElem`的类型表明,只要您可以确定某个类型`ty`中的值是否相等,您就可以确定类型为`ty`的值是否包含在类型为`ty`的向量中。

请记住,`Dec`具有以下构造函数:

是的,它把谓词成立的证明作为它的论据。在这种情况下,对于作为输入传递给`isElem`的值和`xs`中的任何一个,它都是`Elem`值`xs`类型的值。否,它将谓词不成立的证明作为它的论据。在这种情况下,这是一个`Elem value xs -> Void`类型的值。

`isElem`是在`Data.Vect`中定义的,但是看看如何自己编写它是有启发性的。以下清单显示了我们的起点,在名为`Elem Type.idr`的文件中手动定义`Elem`。

清单 9.4 手动定义`Elem`和`isElem` (`ElemType.idr`)

```
data Elem : a -> Vect ka -> 输入 where
  这里 : Elem x (x :: xs)
  那里 : (后来 : Elem x xs) -> Elem x (y :: xs)

isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs)
```

1 定义、细化通过对输入向量`xs`进行大小写拆分来定义函数:

```
isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs) isElem value [] = ?isElem_rhs_1 isElem value (x :: xs) = ?
isElem_rhs_2
```

2 精炼 对于`?isElem_rhs_1`,值显然不在空向量中,因此您可以返回`No`。请记住,`No`将证明您不能构造谓词作为参数。你可以暂时为这个论点留下一个洞:

```
isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs) isElem value [] = No ?notInNil isElem value (x :: xs) = ?
isElem_rhs_2
```

如果你检查`?notInNil`的类型,你会发现要填补这个洞,你需要提供一个证明,证明不能有空向量的元素:

```
a : 类型值 : a
约束 : DecEq a
-----
notInNil : Elem 值 [] -> Void
```

我们很快就会回到这个洞。

3 Refine - 对于`?isElem_rhs_2`,如果`value`和`x`相等,则您已找到所需的值。您可以在结果上使用`decEq`,`case-split`,如果结果是`Yes prf`的形式,则返回`Yes`并带有一个孔,以证明该值是向量中的第一个:

```
isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs)
```

```
isElem value [] = No ?notInNil isElem value (x :: xs) =
case decEq value x of
    是 prf => 是 ?isElem_rhs_1 否 notHere => ?
    isElem_rhs_3
```

4 输入,细化 您可能想用Here填充?isElem_rhs_1孔,但如果您检查它的类型,您会发现您还不能完全使用Here :

```
a:类型值:a
x:a
prf:值 = xk:Nat
xs : 你是 Vect
约束:DecEq a
-----
isElem_rhs_1 : Elem 值 (x :: xs)
```

您不能使用Here ,因为您要查找的类型不是Elem value (value :: xs) 的形式。但是prf告诉你value和x必须相同,所以如果你对prf 进行大小写拆分,你会得到:

```
isElem value (x :: xs) = case decEq value x of
    是 Refl => 是 ?isElem_rhs_2 否 notHere => ?isElem_rhs_3
```

新创建的孔的类型?isElem_rhs_2现在是您需要的形式:

```
a:类型值:a
k : 晚上
xs : 你是 Vect
约束:DecEq a
-----
isElem_rhs_2 : Elem 值 (值 :: xs)
```

您可以使用 Atom 中的表达式搜索来填写?isElem_rhs_2 :

```
isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs) isElem value [] = No ?notInNil isElem value (x :: xs) = case
decEq value x of
```

```
是的 Refl => 是的,这里
没有 notHere => ?isElem_rhs_3
```

5 Refine 对于?isElem_rhs_3, value和x不相等 (你知道这一点是因为decEq value x返回了一个证明) ,所以你可以在xs中递归搜索:

```
isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs) isElem value [] = No ?notInNil isElem value (x :: xs) = case
decEq value x of
```

```
是的 Refl => 是的,这里
No notHere => case isElem value xs of
    是 prf => 是 ?isElem_rhs_1 否 notThere => 否 ?
    isElem_rhs_2
```

表达式搜索将找到?isElem_rhs_1 的必要证明。您现在可以为No case留下一个洞：

```
isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs) isElem value [] = No ?notInNil

isElem value (x :: xs) = case decEq value x of
    是的 Refl => 是的,这里
    No notHere => case isElem value xs of
        是 prf => 是 (有 prf)
        没有 notThere => 没有 ?notInTail
```

在这个阶段,您可以在REPL 中测试定义。如果一个值在一个向量中,你会看到Yes和一个证明:

```
*ElemType> isElem 3 [1,2,3,4,5]
是 (There (There Here)) : Dec (Elem 3 [1, 2, 3, 4, 5])
```

如果没有,你会看到No,有一个洞来证明元素丢失:

```
*ElemType> isElem 7 [1,2,3,4,5]
否 ?notInTail : Dec (Elem 7 [1, 2, 3, 4, 5])
```

要完成定义,您需要完成notInNil和notInTail。

6定义、改进 您可以通过使用 Ctrl-Alt-L 将其提升到顶级函数然后对其参数进行大小写拆分来完成?notInNil。Idris 注意到两种输入都不可能:

```
notInNil : Elem value [] -> Void notInNil 这里不可能 notInNil
(There _) 不可能
```

7定义 您可以类似地完成?notInTail ,但您需要更加努力地证明每种情况都是不可能的。提升到顶级定义,然后对参数进行大小写拆分会导致以下结果:

```
notInTail : (notThere : Elem value xs -> Void) -> (notHere : (value = x) -> Void) -> Elem value (x :: xs) -> Void notInTail notThere notHere Here = ?notInTail_rhs_1 notInTail
notThere notHere (后面有)= ?notInTail_rhs_2
```

对于每个孔,请记住检查其类型和局部变量的类型,因为这些通常会强烈提示如何进行。

8精炼 对于每种情况,您可以使用notHere或notThere来生成您需要的Void类型的值:

```
notInTail : (notThere : Elem value xs -> Void) -> (notHere : (value = x) -> Void) ->
    Elem 值 (x :: xs) -> Void
    notInTail notThere notHere Here = notHere Refl
    notInTail notThere notHere (There later) = notThere later
```

成员资格测试:Elem 谓词

这是完整的定义,供参考。

清单 9.5 isElem (ElemType.idr) 的完整定义

```
notInNil : Elem value [] -> Void notInNil 这里不可能 notInNil (There
_) 不可能

notInTail : (notThere : Elem value xs -> Void) -> (notHere : (value = x) -> Void) -> Elem value (x :: xs) ->
Void
notInTail notThere notHere Here = notHere Refl
notInTail notThere notHere (There later) = notThere later

isElem : DecEq a => (value : a) -> (xs : Vect na) -> Dec (Elem value xs) isElem value [] = No notInNil isElem value (x :: xs)

= case decEq 值 x 的
    是的 Refl => 是的,这里
    No notHere => case isElem value xs of
        是 prf => 是 (有 prf)
        No notThere => No (notInTail notThere notHere)
```

作为比较,清单 9.6 显示了如何定义一个布尔测试来检查向量成员。在这里,我使用了Eq接口,因此我们无法从类型中得到任何关于相等测试行为方式的保证。然而, elem遵循类似于decElem 的结构。

清单 9.6 一个值是否在向量中的布尔测试 (ElemBool.idr)

```
elem : Eq ty => (value : ty) -> (xs : Vect n ty) -> Bool elem value [] = False elem value (x :: xs) = case value == x of
    False => elem 值 xs
    真 => 真
```

这两个定义具有相似的结构,但在isElem中需要做更多的工作来证明不可能的情况确实是不可能的。作为权衡,不需要对isElem进行任何测试,因为类型足够精确,实现必须正确。

练习



- 1 Data.List包含一个Elem for List版本,其工作方式类似于Elem for Vect。
你会如何定义它?
- 2以下谓词表明特定值是列表中的最后一个值:

```
数据最后:列出 a -> a -> 键入 where
LastOne : 最后一个 [value] 值
LastCons : (prf : 最后 xs 值) -> 最后 (x :: xs) 值
```

因此,例如,您可以构造Last [1,2,3] 3 的证明:

```
last123 : 最后 [1,2,3] 3
```

```
last123 = LastCons (LastCons LastOne)
```

编写一个isLast函数来确定一个值是否是List 中的最后一个元素。

它应该具有以下类型：

```
isLast : DecEq a => (xs : List a) -> (value : a) -> Dec (Last xs value)
```

您可以在REPL上测试您的答案,如下所示：

```
*ex_9_1> isLast [1,2,3]
是 (LastCons (LastCons LastOne)) : 十二月 (Last [1, 2, 3] 3)
```

9.2 用类型表示程序状态:猜谜游戏

在实践中,当我们编写在类型中表达系统状态特征 (例如向量长度)的函数时,自然会需要像Elem这样的谓词和像removeElem这样的函数。为了结束本章,我们将看一个发生这种情况的小例子:使用类型系统对猜谜游戏 Hangman 的简单属性进行编码。

在 Hangman 中,玩家尝试通过一次猜测一个字母来猜测一个单词。如果他们猜出单词中的所有字母,他们就赢了。否则,他们被允许进行有限次数的错误猜测,之后他们就输了。

9.2.1 表示游戏的状态

清单 9.7 展示了我们如何将文字游戏的当前状态表示为 Idris 中的数据类型WordState 。状态中有两个重要部分可以捕捉游戏的特征,它们被写成类型的一部分:

玩家剩余的猜测次数
玩家仍然必须猜测的字母数量

清单 9.7 游戏状态 (Hangman.idr)

```
数据 WordState : (guesses_remaining : Nat) -> (letters : Nat) -> 键入 where
```

```
    MkWordState : (word : String) ->
```

```
        (缺少 :Vect 字母字符) ->
```

```
        WordState guesses_remaining letters
```

玩家试图猜测的单词

单词中尚未正确猜到的字母

guesses_remaining 是
MkWordState 的一个隐含参数,我们
将在类型中跟踪它。

如果玩家剩余的猜测数为零 (在这种情况下玩家输了)或剩余的字母数为零 (在这种情况下玩家赢了),则游戏结束。清单 9.8 展示了我们如何在数据类型中表示它。我们可以确定这仅捕获了赢或输的游戏,因为我们将猜测次数和剩余字母数作为WordState 的参数。

清单 9.8 精确定义游戏何时处于完成状态 (Hangman.idr)

数据完成:输入位置

丢失: (游戏:WordState 0 (S 字母)) -> 完成
 赢: (游戏:WordState (S 猜测)) -> 完成

如果剩下的猜测为零,
则游戏失败。

如果零个字母则赢得游戏
还有待猜测。

WordState 依赖类型存储游戏所需的核心数据。通过将规则的核心组成部分 猜测的数量和字母的数量 作为WordState 的参数,我们将能够准确地看到任何函数如何

使用一个WordState 来实现游戏规则。

9.2.2 顶级游戏功能

我们将采用自上而下的方法,定义实现完整游戏的顶级函数。下面的清单给出了起点。

清单 9.9 顶级游戏函数 (Hangman.idr)

game : WordState (S guesses) (S letters) -> IO Finished
 游戏 st = ?game_rhs
 ?game_rhs 需要读取一个字母并根据猜测更新游戏状态。

游戏类型表明如果至少还有一个猜测,游戏可以继续进行
 (S 猜测)并且至少还有一个字母要猜测 (S 字母)。它返回IO 完成,
 这意味着它执行交互式操作,在Finished中生成游戏数据
 状态。

要实现游戏,您需要阅读用户的一封信 (任何
 其他输入无效),检查用户输入的字母是否在目标词中
 在游戏状态下,更新游戏状态,如果游戏未完成则循环。你可以
 通过以下方式之一更新状态:

字母在单词中,在这种情况下你继续游戏
 猜测次数和少一个字母。
 字母不在单词中,在这种情况下你继续游戏
 更少的猜测和相同数量的字母。

使用和剩余的猜测和字母的数量明确记录在
 游戏状态的类型。因此,下一步是编写一个函数来捕获这些
 状态更新其类型。

9.2.3 验证用户输入的谓词:ValidInput

您可以使用IO 操作getLine 读取用户输入:

getLine : IO 字符串

但是因为用户在猜测一个字母,所以唯一有效的输入是那些只包含一个字符的输入。为了精确起见,您可以在谓词中明确表示有效输入的概念。因为String是一个原语,所以很难推断出它在一个类型中的各个组件,因此您可以使用List Char来表示谓词中的输入并根据需要进行转换:

```
data ValidInput : List Char -> Type where
    字母 : (c : Char) -> ValidInput [c]
```

要检查字符串是否为有效输入,您可以编写以下函数,该函数使用Dec返回输入有效的证明或永远不会有效的证明:

```
isValidString : (s : String) -> Dec (ValidInput (unpack s))
```

您将在 9.2.5 节中很快看到isValidString的定义。作为练习,您可以尝试自己编写它,从以下辅助函数开始:

```
isValidInput : (cs : List Char) -> Dec (ValidInput cs)
```

然后,不使用getLine,而是编写一个返回用户猜测的readGuess函数,以及一个保证用户猜测是有效输入的谓词实例:

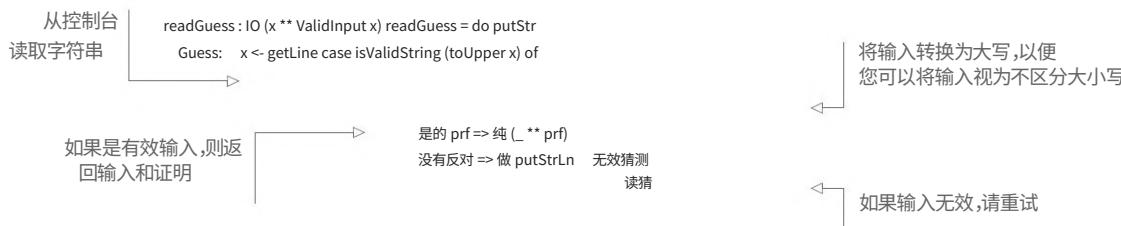
```
readGuess : IO (x ** ValidInput x)
```

readGuess返回一个依赖对,其中第一个元素是从控制台读取的输入,第二个元素是输入必须满足的谓词。

返回值的契约通过返回一个与该值的谓词配对的值, readGuess的类型表示返回值必须满足的契约。

下面的清单给出了readGuess 的定义。

**清单 9.10 从控制台读取猜测,保证有效
(刽子手.idr)**



通过使用readGuess,您可以确定从控制台读取的任何有效字符串都保证只包含一个字符。

9.2.4 处理猜测

如果猜测正确,处理猜测将返回具有一种类型的游戏状态,如果猜测不正确则返回不同类型。您可以使用 Either 泛型类型来表示它。请记住, Either 是 Prelude 中定义的泛型类型,它带有两种可能类型的值:

数据 要么 ab = 左 a |右 b

无论是通常用于表示可能失败的计算结果,如果它确实失败,则携带来有关错误的信息。按照惯例,我们将 Left 用于错误情况, Right 用于成功情况。¹您可以将错误的猜测视为错误情况,因此下一个清单显示了您将用于处理

猜测。

清单 9.11 处理用户猜测的函数类型 (Hangman.idr)

```
processGuess: (字母:字符) ->
    WordState (S 个猜测) (S 个字母) ->
        要么 (WordState 猜测 (S 字母) )
            (WordState (S 猜测)字母)
    输入游戏状态,至少剩下一个猜测和
    至少一个要猜测的字母
    正确猜测 (Right
    case) ,所以去掉一
    个字母来猜测
    猜错了
    (左边的情况),所以删
    除一个可用的猜测
```

这种类型表明,给定一个字母和一个输入游戏状态,它将产生一个新的游戏状态,其中剩余的猜测数量减少(猜测不正确),或者剩余的字母数量减少(猜测是正确的。)

作为抽象状态的类型 WordState 类型的值猜测字母包含有关系系统状态的具体信息,包括要猜测的确切单词以及仍然缺少哪些字母。类型本身表达了关于游戏状态的抽象信息(猜测剩余和丢失字母的数量),它允许您在函数类型中表达规则,如 processGuess。

您可以按如下方式实现 processGuess:

1 定义 你需要检查输入的游戏状态,所以你可以在
输入状态:

```
processGuess: (字母:字符) -> WordState (S 猜测) (S 字
母) -> Either (WordState 猜测 (S 字母) )
```

```
(WordState (S 猜测)字母) processGuess 字母
(MkWordState 单词缺失) = ?guess_rhs_1
```

¹ “正确”也是“正确”的同义词。

2定义 如果猜到的字母是正确的,它将在缺失的向量中,您需要将其删除。您可以使用isElem来查找它是否存在并定义

对isElem 的结果进行大小写拆分:

```
processGuess 字母 (MkWordState 单词缺失)= case isElem 字母缺失
```

```
    是 prf => ?guess_rhs_2 没有相反 => ?
    guess_rhs_3
```

3优化如果字母在缺失字母的向量中,您将返回一个新状态,其中要猜测的字母减少。否则,您将返回一个新状态,可用的猜测更少:

```
processGuess 字母 (MkWordState 单词缺失)= case isElem 字母缺失
```

```
    Yes prf => Right (MkWordState word ?nextVect)
    没有 contra => Left (缺少 MkWordState 字)
```

你有一个洞, ?nextVect,用于丢失字母的更新向量。

4类型,细化 ?nextVect的类型表明您需要删除一个元素从向量:

```
字:字符串 字母:字符
字母:Nat
缺失:Vect (S 字母) Char prf:Elem 字母缺失猜测:Nat
```

nextVect : Vect 字母字符

您可以通过使用removeElem删除缺失的字母来完成定义,并且 Idris 将找到必要的证明prf,作为自动隐式。这样就完成了定义:

```
processGuess: (字母:字符) ->
  WordState (S 个猜测) (S 个字母) ->
  要么 (WordState 猜测 (S 字母) )
  (WordState (Sguess) letters) processGuess letter
  (MkWordState word missing) = case isElem letter missing of

    Yes prf => Right (MkWordState 字 (removeElem 字母缺失))
    没有 contra => Left (缺少 MkWordState 字)
```

显式假设在游戏状态下,您假设仍要猜测的字母数与缺失字母向量的长度相同。通过把它放在 WordState 的类型和processGuess的类型中,你可以确定如果你违反了这个假设,你的程序将不再编译。

可以使用processGuess完成游戏的定义,更新游戏状态
有必要的。

9.2.5 判断输入有效性:检查ValidInput

在完成游戏实现之前,您需要完成isValidString的实现,它决定用户输入的字符串是否是有效输入。

清单 9.12 显示输入字符串必须是有效或无效输入 (刽子手.idr)

```
data ValidInput : List Char -> Type where
    字母 : (c : Char) -> ValidInput [c]

isValidNil : ValidInput [] -> Void isValidNil (字母_) 不可能
    ← 一个字符的输入是有效的。
    ← 请记住 [c] 对 c :: [] 进行了脱糖。

isValidTwo : ValidInput (x :: y :: xs) -> Void isValidTwo (Letter _) 不可能
    ← ValidInput 的构造函数都没有 ValidInput []
        类型。

isValidInput : (cs : List Char) -> Dec (ValidInput cs) isValidInput [] = No
    ← isValidNil
    ← isValidInput (x :: []) = Yes (字
        母 x)
    ← isValidInput (x :: (y :: xs)) = No
    ← isValidTwo

isValidString : (s : String) -> Dec (ValidInput (unpack s)) isValidString s = isValidInput (unpack s)
```

← 唯一有效的情况,因为
构造函数
ValidInput 的类型
为 ValidInput (x ::
[])。

您可以在REPL中测试此定义,方法是使用>>=获取readGuess的输出,匹配依赖对的第一个组件,并将其传递给println:

```
*Hangman> :exec readGuess >>= \(x ** _) => println x
猜测:坏人
猜测无效
猜测:
猜测无效
猜测:f
[ F ]
```

← 由用户输入
 ← 由 readGuess 放置
 ← 由用户输入
 ← 由 readGuess 输出
 ← 由用户输入
 ← 由 readGuess 输出

现在您已经能够读取保证为有效形式的输入,并且能够处理猜测以更新游戏状态,您可以完成顶级游戏实现。

9.2.6 完成顶级游戏实现

下一个清单显示了完成游戏实现的一种可能(如果是基本的)方式。它使用processGuess检查用户的输入并报告猜测是否正确,直到玩家赢或输。

清单 9.13 顶级游戏函数 (Hangman.idr)

您需要检查猜测和信件以确定玩家是赢还是输,因此将它们纳入范围。

通过对 ValidInput 谓词进行模式匹配来提取字母。



猜测是错误的;检查是否有是否还有任何猜测,如果有,请继续。



猜测是正确的;检查是否有剩余字母,如果有则继续。

通过包括猜测的数量和类型中剩余的字母数量,你基本上已经在猜测函数的类型中编写了一个重要的游戏规则。因此,实现中的某些类型的错误不会发生。为了例如,只要你总是使用guess函数来更新游戏状态,你避免以下潜在问题:

在继续游戏之前不可能测试错误的变量(猜测或字母)²这样做会导致类型错误。

不可能继续一个已完成的游戏,因为必须至少有一个剩下的猜测和至少一个要猜测的字母。当两个字母都丢失时,过早完成游戏是不可能的并且猜测仍然可用。

以下清单显示了设置游戏的main的可能实现带有目标词“测试”,因此要求玩家猜测字母“T”、“E”和“S”。

清单 9.14 设置游戏的主程序 (Hangman.idr)

设置一个允许猜错2次的新游戏和一个目标词“测试”

² 我在写这个例子的时候确实犯了这个错误!

9.3 总结

你可以编写类型来表达关于值如何关联的假设。Elem 依赖类型是一个谓词,表示一个值必须包含在一个向量中。

通过将谓词作为参数传递给函数,您可以表达合同
函数的输入必须遵循。

你可以编写一个函数来显示一个谓词是可判定的,使用Dec。Idris 将尝试查
找由表达式标记为auto的参数的值
搜索。

您可以在
类型。

谓词可以描述用户输入的有效性,保证用户输入
必要时进行验证。

视图:扩展模式 匹配

本章涵盖

定义视图,描述不同形式的
模式匹配

介绍使用视图的with构造描述数据结构的有效遍历隐藏视图后面
的数据表示

在类型驱动的开发中,我们实现函数的方法是编写一个类型,通过对其参数的大小写拆分来定义函数的结构,并通过填充漏洞来细化函数。特别是在定义步骤中,我们使用输入类型的结构来驱动整个函数的结构。

正如您在第3章中所了解的,当您对变量进行大小写拆分时,模式来自变量的类型。具体来说,模式来自可用于构建该类型值的数据构造函数。例如,如果您对List elem类型的items变量进行大小写拆分,则会出现以下模式:

[]表示空列表 (x :: xs)表示一个非
空列表,其中x作为第一个元素, xs作为其余元素的列表

因此,模式匹配将变量解构为它们的组件。但是,您通常会希望以不同的方式解构变量。例如,您可能希望将输入列表解构为以下形式之一: 除了最后一个元素之外的所有元素的列表,然后是最后一个元素两个子列表
输入的前半部分和后半部分

在本章中,您将看到如何通过定义信息数据类型 (称为视图) 来扩展您可以使用的模式形式。视图是由您想要匹配的数据参数化的依赖类型,它们为您提供了观察该数据的新方法。例如,视图可以执行以下操作:

描述新形式的模式,例如允许您匹配列表的最后一个元素而不是第一个元素

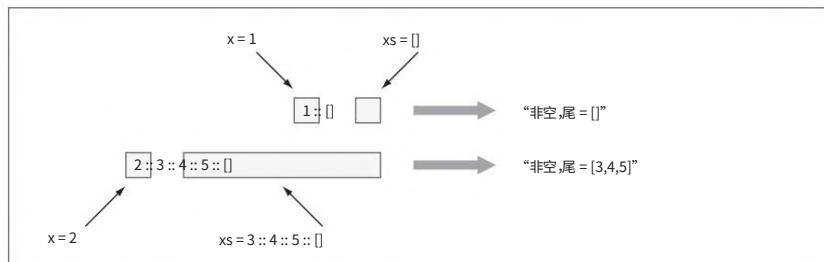
定义遍历数据结构的替代方法,例如反向遍历列表,或重复将列表分成两半将复杂的数据表示隐藏在抽象接口后面,同时仍支持对该数据进行模式匹配

我们将首先了解如何使用视图定义列表匹配的替代方法,例如首先处理最后一个元素。然后,我们将研究定义有效的列表遍历并保证它们终止,我们将研究 Idris 库提供的一些遍历。最后,我们将使用视图通过将数据结构隐藏在单独的模块中并使用视图匹配和遍历数据来帮助进行数据抽象。

10.1 定义和使用视图

当你编写一个($x :: xs$)模式来匹配一个列表时, x 将获取列表的第一个元素的值, xs 将获取列表尾部的值。你看到了 fol

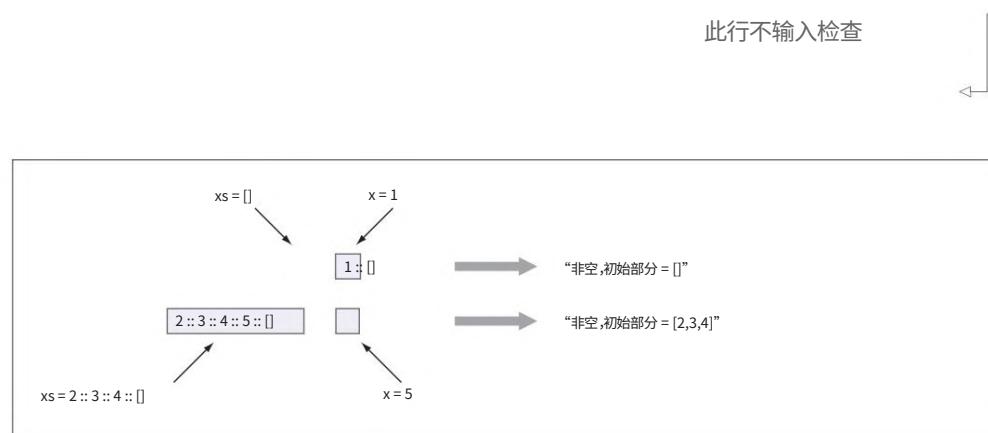
第 3 章中的 lowing 函数来描述列表是如何构建的,如图 10.1 所示:



因此,使用模式($x :: xs$)匹配列表意味着您将始终向前遍历列表,首先处理 x ,然后处理尾部 xs 。但有时能够方便地向后遍历列表,首先处理最后一个元素。

您可以使用 $++$ 运算符将单个元素添加到列表的末尾:

能够匹配($xs ++ [x]$)形式的模式会很好,其中 x 获取列表最后一个元素的值, xs 获取列表初始段的值。也就是说,您可能希望能够编写如下形式的函数,如图 10.2 所示:



不幸的是,如果您尝试实现`describeListEnd`,它将失败并显示以下内容

错误:

问题是您正在尝试对函数的结果进行模式匹配, $++$,并且通常 Idris 无法从其输出中自动推断出任意函数的输入。但是,您可以通过定义称为视图的信息数据类型来扩展您可以使用的模式的形式。在本节中,您将看到如何定义视图,并且我将介绍`with`构造,它为定义使用视图的函数提供了一种简洁的符号。

10.1.1 匹配列表中的最后一项

不能直接写`describeListEnd`因为不能直接匹配($xs ++ [x]$)形式的模式。但是,您可以利用依赖模式匹配来推断特定输入必须具有($xs ++ [x]$)形式。你看到了

第 6 章中的依赖模式匹配,其中检查一个参数的值

(即对该论点进行大小写拆分)可以告诉您另一个的形式。使用
依赖模式匹配,您可以描述列表的替代模式。

下面的清单显示了ListLast依赖类型,它描述了List可以采用的两种可能形式。一个列表要么是空的, [],
要么是从初始的
列表的一部分及其最后一个元素。

**清单 10.1 ListLast依赖类型,它提供了列表的替代模式
(描述列表.idr)**

ListLast : List a -> Type where

一个空列表有数据
表格 []。
非空列表有一个首字母
部分 xs 和最后一项 x。

使用ListLast以及依赖模式匹配,您可以定义描述 ListEnd。您将首先定义一个辅助函数,该函数接受输入列
表、输入和

一个额外的参数form,表示列表是空的还是非空的:

1类型 - 从类型开始。您将使用ListLast来描述可能的形式
该输入可以采用:

2定义 你想描述输入,所以你需要以某种方式检查它的形式。
以前,当定义一个函数来检查输入的形式时,你已经
对该输入进行大小写拆分。在这里, form参数的目的是告诉你
更多关于输入的信息,所以改为区分大小写:

请注意,表单上的案例拆分已经告诉您更多关于输入形式的信息。在
特别是, NonEmpty的类型意味着如果form的值为NonEmpty xs x,
那么输入必须具有值 (xs ++ [x])。

3细化 通过描述我们最初的形式来完成定义
尝试describeListEnd:

ListLast是列表视图,因为它提供了另一种查看数据的方法。
不过,它是一种普通的数据类型,为了在实践中使用ListLast,您将
需要能够将输入列表xs转换为ListLast xs类型的值。

10.1.2 构建视图:覆盖功能

下一个清单显示listLast,它将输入列表xs转换为视图ListLast xs,它允许访问xs 的最后一个元素。

清单 10.2以ListLast (DescribeList.idr)的形式描述一个列表

您需要遍历整个列表才能找到最后一个元素,因此进行递归调用。

total 标志意味着如果 listLast 未完全定义,Idris 将报告错误,确保 listLast 将适用于每个列表。

listLast是ListLast视图的覆盖函数。视图的覆盖函数描述如何将一个值 (在本例中为输入列表)转换为该值的视图 (在这种情况下,一个xs列表和一个值x,其中 $xs \text{ ++ } [x] = \text{输入}$)。

覆盖函数的命名约定按照惯例,覆盖函数的名称与视图类型相同,但首字母小写信。

既然可以用ListLast的形式描述任何List ,就可以完成describeListEnd 的定义:

这正如我们在最初尝试describeListEnd 时所预期的那样工作,使用原始 describeHelper 中的模式。鉴于您使用的是不同的模式概念比默认匹配,您应该期望必须使用一些额外的符号解释如何匹配模式 ($xs \text{ ++ } [x]$) ,但总体定义还是感觉相当冗长。

因为这种方式的依赖模式匹配是一种常见的编程方式 Idris 中的成语,有一个用于表达扩展模式匹配的结构简而言之: with构造。

10.1.3 with blocks:扩展模式匹配的语法

使用视图生成信息模式,如($xs \text{ ++ } [x]$)有助于提高可读性函数并增加您对其正确性的信心,因为类型告诉您究竟是什么形式的输入必须采取。但是这些功能可以多一点冗长,因为您需要创建一个额外的辅助函数 (如describeHelper)来

做必要的模式匹配。with构造提供了使用的符号
视图更简洁。

使用with块,您可以在定义的左侧添加额外的参数,从而为您提供更多参数来区分大小写。了解其工作原理的最简单方法是通过示例,因此让我们看一下如何使用with块来定义describeListEnd:

1种类型 从一个顶级类型开始,它接受一个List Int并返回一个字符串:

```
describeListEnd : 列表 Int -> 字符串
describeListEnd 输入 = ?describeListEnd_rhs
```

2定义 - 按 Ctrl-Alt-W,光标位于孔?describeListEnd_rhs 的行上。这将添加一个新的模式子句,如下所示 (它还不会进行类型检查) :

```
describeListEnd : 列表 Int -> 字符串
describeListEnd 输入与 (_)
    describeListEnd 输入 | with_pat = ?describeListEnd_rhs
```

3定义with语法 在定义的左侧添加一个新参数。

with后面的括号给出该参数的值,以及 (缩进)

下面的子句在竖线之后包含额外的参数。在这里,您将

匹配listLast 输入的结果,因此替换

使用listLast 输入:

```
describeListEnd : 列表 Int -> 字符串
describeListEnd 输入与 (listLast 输入)
    describeListEnd 输入 | with_pat = ?describeListEnd_rhs
```

4类型 如果您检查?describeListEnd_rhs 的类型,您可以看到更多详细信息

关于with的工作原理,具体看with_pat 的类型:

```
输入: 列表整数
with_pat : ListLast 输入
```

describeListEnd_rhs : 字符串

with_pat的值是listLast输入的结果。图 10.3 显示了

with构造的语法组成部分。请注意,范围

with块由缩进管理。

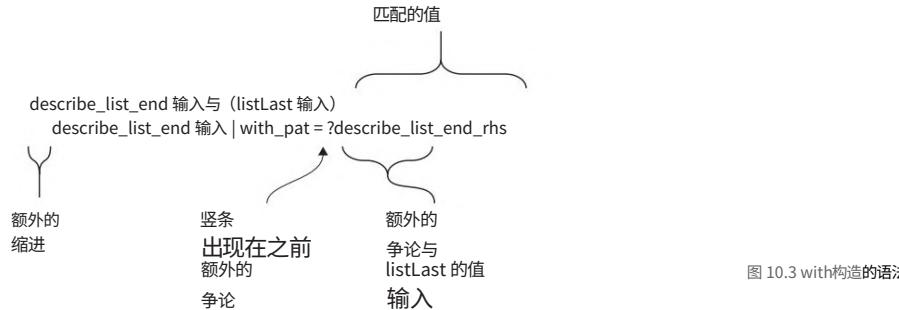


图 10.3 with 构造的语法

5定义 注意input和with_pat的类型与之前describeHelper的输入类型完全相同。和describeHelper的定义一样，如果您在with_pat上进行大小写拆分，您将了解有关输入形式的更多信息：

6细化 - 像以前一样完成定义，并带有模式的描述：

实际上，with构造允许您在listLast输入的结果上使用中间模式匹配，而无需定义像describeHelper这样的单独函数。反过来，匹配listLast输入的结果为您提供更多信息丰富的输入模式。

WITH 和 CASE之间的区别with构造的目的与case块的目的相似，因为它允许匹配中间结果。然而，有一个重要的区别：with引入了一个新的模式来匹配定义的左侧。因此，您可以直接使用依赖模式匹配。以describeListEnd为例，listLast输入的结果上的匹配告诉你输入必须采取的形式。

您不能在模式中只使用任何表达式，因为一般来说，仅给定函数的结果就不可能决定函数的输入必须是什么。因此，Idris仅在可以推断出这些输入时才允许模式，在以下情况下：

该模式由应用于某些参数的数据构造函数组成。参数也必须是有效的模式。模式的值是由其他一些有效模式强制的。在describeListEnd的情况下，模式(xs ++ [x])的值由有效模式NonEmpty xs x 强制。

10.1.4示例:使用视图反转列表

一旦您能够使用视图以不同的方式进行模式匹配，您就可以以新的方式遍历数据结构。例如，您可以使用listLast从右到左遍历列表，而不是总是从左到右遍历列表，首先检查最后一个元素。

你可以这样反转一个列表：要反转一个空列表[],返回[]。要反转xs ++ [x]形式的列表，反转xs然后将x添加到前面名单。

您可以使用ListLast视图直接实现此算法：

1类型 调用函数myReverse因为已经有一个反向函数
在前奏曲中：

```
myReverse : List a -> List a
myReverse _ = ?myReverse_rhs
```

2定义 - 您将通过首先检查输入的最后一个元素来定义函数，因此您可以使用listLast来匹配ListLast 输入类型的值。
按 Ctrl-Alt-W 插入一个with块，并添加listLast 输入作为要检查的值：

```
myReverse : List a -> List a
myReverse input
with (listLast input)
  我的反向输入 | with_pat = ?myReverse_rhs
```

3定义 接下来，对with_pat进行大小写拆分，以提供相关的输入模式：

```
myReverse : List a -> List a
myReverse input
with (listLast input)
  我的反向 [] | 空 = ?myReverse_rhs_1 myReverse (xs ++ [x]) |
  (NonEmpty xs x) = ?myReverse_rhs_2
```

4细化最后，您可以完成如下定义：

```
myReverse : List a -> List a
myReverse input
with (listLast input)
  [] = myReverse (xs ++ [x])
  (NonEmpty xs x) = x :: myReverse xs
```

这是算法的一个相当直接的实现，反向遍历列表并通过将输入的最后一项添加为结果的第一项来构造一个新列表。然而，存在两个问题：定义效率低，因为它在每次递归调用时都构造 ListLast 输入，并且构造ListLast输入需要遍历输入。Idris 无法确定定义是否是完全的：

```
*Reverse> :total myReverse Main.myReverse
可能不完全,原因是:
由于递归路径,可能不完全:在 Main.myReverse 中有块,在 Main.myReverse
中有块
```

有关 Idris 中的整体检查的简要讨论，请参见侧边栏。

第一个问题很重要，因为可以在线性时间内编写myReverse，只遍历输入列表一次。从类型驱动开发的角度来看，第二个问题很重要：正如我在第 1 章中所讨论的，如果 Idris 可以确定一个函数是完全的，那么您就可以有力地保证类型准确地描述了该函数将要做什么。如果没有，您只能保证函数在没有崩溃的情况下终止时将产生给定类型的值。此外，如果 Idris 不能确定一个函数是全数的，它也不能确定任何调用它的函数是全数的。

我们将研究如何解决这些问题,只需对 myReverse 本身,在第 10.2 节中。

整体检查

Idris 试图通过检查两件事来确定函数是否总是终止:

所有可能的类型良好的输入都必须有模式。

当有递归调用 (或一系列相互递归调用) 时,
必须有一个趋于基本情况的递减论点。

为了确定哪些参数正在减少,伊德里斯查看了
定义中的输入。如果模式采用数据构造函数的形式,Idris 认为
该模式中的参数小于输入。例如,在 myReverse 中,Idris 不认为 $xs < (xs ++ [x])$,因为 $(xs ++$
 $[x])$ 不是数据构造函数的形式。

此限制使 Idris 可以检查减少参数的概念。

一般来说,Idris 无法判断函数的输入是否总是小于
结果。

正如您将在 10.2 节中看到的,您可以通过定义递归视图来解决这个限制。

10.1.5 示例:归并排序

视图允许您以任何您喜欢的方式描述数据结构上的匹配,有尽可能多的
只要您可以为视图实现覆盖功能,就可以随意使用模式。作为
第二个视图示例,我们将在 Lists 上实现归并排序算法。

归并排序在高层次上的工作原理如下:

如果输入是一个空列表,则返回一个空列表。

如果输入有一个元素,它已经排序,所以返回输入。

在所有其他情况下,将列表分成两半(大小最多相差一个),

递归地对这些部分进行排序,然后将两个已排序的部分合并为一个已排序的
列表(见图 10.4)。

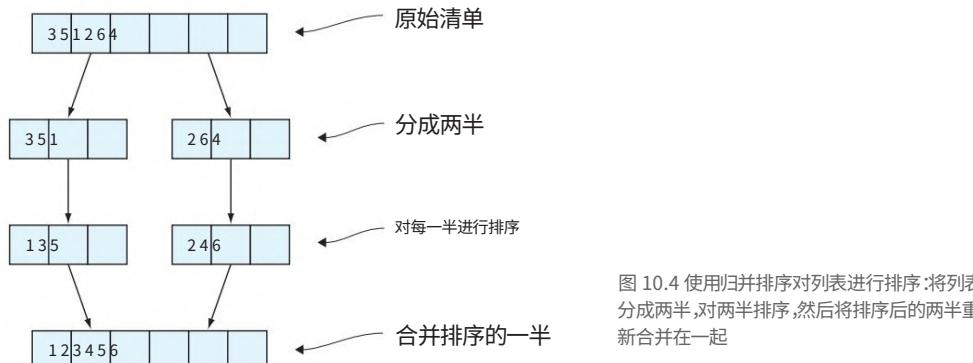


图 10.4 使用归并排序对列表进行排序:将列表
分成两半,对两半排序,然后将排序后的两半重
新合并在一起

如果您有两个排序列表,您可以使用合并功能将它们合并在一起
Prelude 中定义:

```
伊德里斯> :doc 合并
Prelude.List.merge : Ord a => List a -> List a -> List a
    使用类型的默认排序合并两个排序列表
    他们的元素。
```

请注意,它具有泛型类型,并且需要实现Ord接口
列表的元素类型。

假设两个输入列表是排序的, merge将产生一个排序列表
输入列表中的元素。例如,图 10.4 中的列表将被合并为
如下:

```
伊德里斯> 合并 [1,3,5] [2,4,6]
[1, 2, 3, 4, 5, 6] :列表整数
```

清单10.3展示了您可能希望如何编写一个对
列表使用归并排序算法。不幸的是,就目前而言,这行不通
因为(lefts ++ rights)不是有效模式。

清单 10.3使用无效模式进行合并排序的初始尝试(MergeSort.idr)

```
mergeSort : Ord a => List a -> List a
合并排序 [] = []
    ◀————— 一个空列表已经排序。
合并排序 [x] = [x]
    ◀————— 单例列表已经排序。
→ 合并排序 (左 ++ 右)
    = 合并 (合并排序左) (合并排序右)
    ◀————— 递归地对列表的左右两半进行排
    序,然后将结果合并成一个完整的
    排序列表
此模式无效,因为++不是数据构造函数,但您想提
取输入的左半部分和右半部分。
```

给定一个与模式匹配的输入列表, (lefts ++ rights) , Idris 一般不能推断出lefts和rights必须是什么;实际上,如果输入列表具有一个或多个元素,则可能存在几种合理的可能性。这里有几个

例子:

- [1]可以匹配 (lefts ++ rights) , lefts为[1] , rights为[],
或左为[] ,右为[1]。
- [1,2,3]可以匹配(lefts ++ rights)与lefts作为[1]和right
为[2,3],或左为[1,2] ,右为[3],以及许多其他可能性。

为了匹配我们想要的模式,如清单 10.3 所示,您需要创建一个
列表, SplitList,它给出了你想要的模式。下面的清单显示了SplitList的定义,并给出了它的覆盖函数splitList 的类型。

清单 10.4 一个列表视图,它给出了空列表、单例列表和
连接列表 (MergeSort.idr)

```
数据SplitList:列出一个->类型哪里
    拆分无:拆分列表 []
    SplitOne:拆分列表 [x]
    SplitPair : (lefts : List a) -> (rights : List a) ->
        SplitList (左++右)

splitList : (输入 : List a) -> SplitList 输入
```

我将很快给出覆盖函数splitList的定义。现在,请注意,作为
只要splitList的实现是完全的,你可以从它的类型确定它
给出空列表、单例列表或两个列表连接的有效模式。你
但是,对于列表如何拆分成对的类型没有任何保证;在
在这种情况下,您需要依靠实现来确保左右大小最多相差一个。

SPLITPAIR的精度原则,您可以制作SplitPair的类型
更精确,并带有左右大小不同的证明
大多数。实际上,Idris 库模块Data.List.Views导出了这样一个
视图,称为SplitBalanced。

您可以使用SplitList视图实现mergeSort ,如下所示:

1类型 - 从类型和骨架定义开始:

```
mergeSort : Ord a => List a -> List a
合并排序输入 = ?mergeSort_rhs
```

2定义要获得所需的模式,您需要使用SplitList视图

你刚刚创建。添加一个with块并使用splitList覆盖函数:

```
mergeSort : Ord a => List a -> List a
与 (splitList 输入)合并排序输入
    合并排序输入 | with _pat = ?mergeSort_rhs
```

3定义 如果你在with_pat 上区分大小写,你会得到合适的输入模式
由SplitList的构造函数类型引起:

```
mergeSort : Ord a => List a -> List a
与 (splitList 输入)合并排序输入
    合并排序 [] | SplitNil = ?mergeSort_rhs_1
    合并排序 [x] | SplitOne = ?mergeSort_rhs_2
    合并排序 (左 ++ 右) | (SplitPair left right) = ?mergeSort_rhs_3
```

4细化您可以通过填写每个的右侧来完成定义
模式,直接遵循我在本节开头给出的合并排序算法的高级描述:

```
mergeSort : Ord a => List a -> List a
与 (splitList 输入)合并排序输入
    合并排序 [] | SplitNil = []
    合并排序 [x] | SplitOne = [x]
```

定义和使用视图

合并排序 (左 ++ 右) | (SplitPair 向左向右)
= 合并 (合并排序左) (合并排序权限)

在测试mergeSort之前,您需要实现覆盖函数

拆分列表。下一个清单给出了splitList的定义,它返回一个描述空列表模式、单例列表模式或由两个列表串联组成的模式,其中这些列表的长度最多相差一个。

清单 10.5 为SplitList (MergeSort.idr) 定义一个覆盖函数



通过使用对

输入列表作为辅助函数splitListHelp中的计数器,如下所示:

在每次递归调用中,计数器都会遍历列表的两个元素。这

pattern (_ :: _ :: counter) 匹配任何包含至少两个元素的列表,其中
counter是一个包含除前两个元素之外的所有元素的列表。

当计数器到达列表末尾时(即少于两个元素
仍然),您必须遍历输入的一半。

您现在可以在REPL中尝试splitList和mergeSort :

```

*MergeSort> splitList [1]
SplitOne:拆分列表 [1]

*MergeSort> splitList [1,2,3,4,5]
SplitPair [1, 2] [3, 4, 5] : SplitList [1, 2, 3, 4, 5]

*MergeSort> 合并排序 [3,2,1]
[1, 2, 3] : 列出整数

*MergeSort> 合并排序 [5,1,4,3,2,6,8,7,9]
[1, 2, 3, 4, 5, 6, 7, 8, 9] : 列表整数

```

通过定义一个视图,根据如何将输入列表分成两半来给出输入列表的可能情况,您可以编写一个直接实现算法高级描述的 mergeSort 定义。

然而,与 myReverse 一样,Idris 无法判断 mergeSort 是否为总计:

```
*MergeSort> :total mergeSort Main.mergeSort 可能
不完全,原因是:
由于递归路径,可能不是全部:在 Main.mergeSort 中有块,在 Main.mergeSort 中有
块
```

同样,问题在于 Idris 无法确定递归调用肯定位于比原始输入更小的列表中。在下一节中,您将看到如何通过定义描述函数递归结构的递归视图以及定义函数的模式来解决这个问题。

练习



1 TakeN 视图允许一次遍历列表中的多个元素:

```
数据 TakeN : 列出一个 -> 键入 where
更少:TakeN xs
确切的: (n_xs:列表 a) -> TakeN (n_xs ++ rest)
```

```
takeN : (n : Nat) -> (xs : List a) -> TakeN xs
```

Fewer 构造函数涵盖了元素少于 n 的情况。

定义覆盖函数 takeN。

要检查您的定义是否有效,您应该能够运行以下函数,该函数将列表分组为具有 n 个元素的子列表:

```
groupByN : (n : Nat) -> (xs : List a) -> List (List a)
groupByN n xs with (takeN n xs) groupByN n xs | 更少
= [xs] groupByN n (n_xs ++ rest) | (确切的 n_xs)= n_xs :: groupByN n 休息
```

这是一个例子:

```
*ex_10_1> groupByN 3 [1..10]
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10]] : 列表 (列表整数)
```

2 使用 TakeN 定义一个函数,通过计算列表将列表分成两半

长度:

```
一半:列表 a -> (列表 a,列表 a)
```

如果您已正确实施此操作,您应该会看到以下内容:

```
*ex_10_1> 一半 [1..10]
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]] : (List Integer, List Integer)
```

```
*ex_10_1> 一半 [1]
[], [1] : (整数列表,整数列表)
```

提示:使用 div 来划分 Nat。

10.2 递归视图:终止和效率

视图的目的是为我们提供匹配数据的新方法,使用with构造以获得更简洁的语法。当你编写一个带有视图的函数时,你会使用以下组件: 原始输入数据输入数据的视图,其中视图数据类型由输入参数化

数据

视图的覆盖函数,它为视图构造视图的实例

输入数据

然后,使用with构造,您可以在视图上进行模式匹配。相关模式匹配为您提供原始数据的信息模式。您在上一节中已经看到了这两个示例:反转列表,以及将列表分成两半以进行合并排序。然而,在这两种情况下,伊德里斯都无法判断得到的函数是完全的。此外,当您反转列表时,生成的函数效率低下,因为它必须在每次递归调用时重建视图。

在本节中,我们将看看如何通过定义递归视图来描述数据结构的遍历来解决这两个问题。此外,一旦我们定义了一个视图,我们就可以将它重用于任何使用相同递归模式的函数。Idris 库为遍历数据结构提供了许多有用的视图,我们将使用其中的一些来查看一些示例函数。不过,首先,我们将改进myReverse 的定义。

10.2.1 “Snoc”列表:反向遍历列表

snoc列表是将元素添加到列表末尾而不是开头的列表。我们可以将它们定义如下,作为通用数据类型:

数据 SnocList ty = 空 | Snoc (SnocList ty) ty

SNOC 列表术语名称snoc 列表的出现是因为将元素添加到列表开头的运算符的传统名称 (起源于 Lisp)是cons。因此,将元素添加到列表末尾的运算符的名称是snoc。

使用SnocList,您以相反的顺序遍历元素,因为元素 (ty 类型)出现在列表 (SnocList ty 类型)之后。您可以轻松地从元素顺序相反的SnocList生成List :

```
reverseSnoc : SnocList ty -> 列表 ty
reverseSnoc Empty = []
reverseSnoc (Snoc xs x) = x :: reverseSnoc xs
```

您可以通过在等效的List 上参数化SnocList来更精确地显示SnocList和List之间的关系。以下清单显示了如何以这种方式定义SnocList。

**清单 10.6 SnocList类型,通过等效的List参数化
(SnocList.idr)**

```
data SnocList : {xs : List a} -> SnocList where
  [] : SnocList []
  Snoc : {rec : SnocList xs} -> SnocList (xs ++ [x])
```

snocList : {xs : List a} -> SnocList xs

Empty 等价于 [] 表示的列表。

给定一个列表 xs,
snocList 构建一个等效于 xs 的 SnocList。

给定一个等效于列表 xs 的 SnocList 和一个隐式值 x,Snoc 构建一个等效于 xs ++ [x] 的 SnocList。

这在结构上与您在上一节中定义的ListLast非常相似。不同之处在于Snoc采用SnocList xs类型的递归参数。

这里, SnocList是一个递归视图,而snocList是它的覆盖函数。我们很快就会谈到 snocList的定义;首先,让我们看看如何使用它来实现myReverse:

1种类型 首先定义一个辅助函数,该函数接受一个列表、输入及其等效项
alent SnocList:

```
myReverseHelper : {input : List a} -> SnocList input -> List a
myReverseHelper input snoc = ?myReverseHelper_rhs
```

2定义 如果您在snoc 上进行大小写拆分,您将获得相应的输入模式:

```
myReverseHelper : {input : List a} -> SnocList input -> List a
myReverseHelper [] Empty = ?myReverseHelper_rhs_1
myReverseHelper (xs ++ [x]) (Snoc rec) = ?myReverseHelper_rhs_2
```

3优化如果输入为空,您将返回空列表:

```
myReverseHelper : {input : List a} -> SnocList input -> List a
myReverseHelper [] Empty = []
myReverseHelper (xs ++ [x]) (Snoc rec) = ?myReverseHelper_rhs_2
```

4 Refine, type否则,您将递归地反转xs并将项目x添加到正面:

```
myReverseHelper : {input : List a} -> SnocList input -> List a
myReverseHelper [] Empty = []
myReverseHelper (xs ++ [x]) (Snoc rec) = x :: myReverseHelper xs ?snocrec
```

递归调用中的第二个参数还有一个漏洞, ?snocrec 。如果你检查它,你会发现你需要代表xs的SnocList :

```
a :类型 xs :列
出一个
录制:SnocList xs
x:a
-----
snocrec : SnocList xs
```

5 Refine 幸运的是,你已经有了一个SnocList xs类型的值rec ,所以你可以直接用它来完成定义:

```
myReverseHelper : (input : List a) -> SnocList input -> List a
myReverseHelper [] Empty = []
myReverseHelper (xs ++ [x]) (Snoc rec) = x :: myReverseHelper xs rec
```

6 定义 最后,您可以通过构建一个SnocList并调用myReverseHelper 来定义myReverse :

```
myReverse : 列出一个 -> 列出一个 myReverse 输入
入 = myReverseHelper 输入 (snocList 输入)
```

你还不能测试这个,因为你还没有实现snocList,但是现在注意这与第 10.1.4 节中使用ListLast 的myReverse的实现有何不同。相似之处在于您通过在输入视图上进行匹配来找到输入列表的模式。不同之处在于视图是递归的,这意味着您不必在每次递归调用时都重新构建视图;您已经可以访问它了。

myReverseHelper的定义是完整的,因为SnocList参数在每次递归调用时都在减少:

```
*SnocList> :total myReverseHelper Main.myReverseHelper
是 Total
```

现在仍然需要实现snocList。只要您可以通过仅遍历列表一次来实现snocList ,您就会拥有一个以线性时间运行的myReverse实现。下面的清单显示了snocList的实现,它只遍历列表一次,使用带有累加器的辅助函数通过一次添加一个元素来构建SnocList 。

清单 10.7 实现覆盖函数snocList

```
snocListHelp : (snoc : SnocList 输入) -> (rest : List a) ->
    SnocList (输入++休息)
→ snocListHelp {input} snoc [] = 重写 snoc 中的 appendNilRightNeutral 输入 snocListHelp {input} snoc (x :: xs) = 重写 snocListHelp
    中的 appendAssociative 输入 [x] xs (Snoc snoc {x}) xs
```

```
snocList : (xs : List a) -> SnocList xs
snocList xs = snocListHelp
Empty xs
```

将一个空列表附加到
代表输入的SnocList

将SnocList初始化为Empty,然后调
用snocListHelp一次添加xs一个
元素

将元素 x 附加到 a
SnocList 表示输入,然后附加剩
余的元素 ,xs

snocList的定义有点棘手,涉及重写构造 (您在第 8 章中看到)以获取正确形式的类型以构建SnocList。

您将使用 Prelude 中的以下函数重写:

274

第10章视图:扩展模式匹配

```
appendNilRightNeutral : (l : List a) -> l ++ [] = l
appendAssociative : (l : List a) -> (c : List a) ->
(r : List a) ->
l ++ (c ++ r) = (l ++ c) ++ r
```

与任何复杂的定义一样,通过用孔替换定义的子表达式并查看这些孔的类型来尝试理解它是一个好主意。在这种情况下,删除重写构造并用孔替换它们也很有用,以查看类型需要如何重写:

```
snocListHelp : SnocList input -> (xs : List a) -> SnocList (input ++ xs) snocListHelp {input} snoc [] = ?rewriteNil snoc snocListHelp
{input} snoc (x :: xs) = ?rewriteCons (snocListHelp (Snoc snoc {x}) xs)
```

您应该看到`rewriteNil`和`rewriteCons`的以下类型:

```
rewriteNil : SnocList Input -> SnocList (Input ++ []) rewriteCons : SnocList ((Input ++ [x]) ++ xs) ->
SnocList (Input ++ x :: xs)
```

好消息是,一旦定义了`snocList`,就可以将它重用于任何需要反向遍历列表的函数。此外,正如您稍后将看到的,Idris库中还定义了一个`SnocList`视图以及其他几个视图。

10.2.2 递归视图和 with 构造

您现在有一个以线性时间运行的`myReverse`实现,因为它遍历列表一次以构建`SnocList`视图,然后遍历`SnocList`视图

一次建立反向列表。您还可以确认 Idris 相信它是总数:

```
*SnocList> :total myReverse Main.myReverse
Total
```

然而,生成的定义并不像之前的`myReverse`定义那样简洁,因为它不使用`with`构造:

```
myReverseHelper : (input : List a) -> SnocList input -> List a
myReverseHelper [] Empty = []
myReverseHelper (xs ++ [x]) (Snoc rec) = x :: myReverseHelper xs rec
```

```
myReverse : 列出一个 -> 列出一个 myReverse 输入 =
myReverseHelper 输入 (snocList 输入)
```

让我们看看如果你尝试这样做会发生什么:

1类型,定义 - 从以下骨架定义开始:

```
myReverse : List a -> List a
myReverse input with
(snocList input)
我的反向输入 | with_pat = ?myReverse_rhs
```

2定义 您可以通过在`with_pat`上的大小写拆分来编写函数,以获得输入的可能模式:

```
myReverse : List a -> List a
myReverse input with
(snocList input)
```

```
我的反向 [] | 空 = ?myReverse_rhs_1 myReverse (xs ++ [x]) | (Snoc
rec) = ?myReverse_rhs_2
```

3 Refine 像以前一样填写右侧:

```
myReverse : List a -> List a
myReverse input
with (snocList input)
    我的反向 [] | 空 = [] myReverse (xs ++ [x]) |
    (Snoc rec) = x :: myReverse xs
```

不幸的是,这调用了顶级reverse 函数,该函数使用snocList 输入重建视图,因此您遇到与以前相同的问题:

```
*SnocList> :total myReverse Main.myReverse
可能不完全,原因是:
    由于递归路径,可能不完全:在 Main.myReverse 中有块,在 Main.myReverse
    中有块
```

4 Refine 相反,当您进行递归调用时,您可以进行调用直接到with块,使用|右侧的符号:

```
myReverse : List a -> List a
myReverse input
with (snocList input)
    我的反向 [] | 空 = [] myReverse (xs ++ [x]) |
    (Snoc rec) = x :: myReverse xs |记录
```

调用myReverse xs | rec递归调用myReverse,但是绕过snocList输入的构造,直接使用rec 。结果定义是完整的,构建输入的SnocList表示,并遍历:

```
*SnocList> :total myReverse Main.myReverse
是 Total
```

这还具有使myReverse在线性时间内运行的效果。

在实践中,当您使用with构造时,Idris 为with块的主体引入了一个新的函数定义,就像您之前手动实现的myReverseHelper的定义一样。

当你写myReverse xs | rec,这相当于在前面的定义中写了myReverseHelper xs rec 。但是通过改用with构造,Idris 为辅助函数生成了一个合适的类型。

通过使用with构造,您可以以不同的方式对数据结构进行模式匹配和遍历,匹配和遍历的结构由视图的类型给出。此外,由于视图本身是数据结构,Idris 可以确定遍历视图的函数是完全的。

10.2.3 遍历多个参数:用块嵌套

当您编写模式匹配定义时,您通常希望同时匹配多个输入。到目前为止,使用with构造,您只匹配了一个值。

但与任何语言结构一样, with块可以嵌套。

要了解它是如何工作的,让我们定义一个isSuffix函数:

```
isSuffix : Eq a => List a -> List a -> Bool
```

如果第一个参数中的列表是

第二个论点。例如:

```
*IsSuffix> isSuffix [7,8,9,10] [1..10]
真.布尔
```

```
*IsSuffix> isSuffix [7,8,9] [1..10]
假.布尔值
```

您可以通过反向遍历两个列表来定义此函数,采用以下方法
脚步:

1 定义 - 从骨架定义开始:

```
isSuffix : Eq a => List a -> List a -> Bool
isSuffix input1 input2 = ?isSuffix_rhs
```

2 定义、细化接下来,使用snocList匹配第一个输入,以便
首先处理最后一个元素:

```
isSuffix : Eq a => List a -> List a -> Bool
isSuffix input1 input2 with (snocList input1)
  isSuffix [] 输入2 | 空 = ?isSuffix_rhs_1
  isSuffix (xs ++ [x]) 输入2 | (Snoc rec) = ?isSuffix_rhs_2
```

您可以将rec重命名为xsrec,以表明它是xs的递归视图

逆转。然后,如果第一个列表为空,它通常是第二个列表的后缀:

```
isSuffix : Eq a => List a -> List a -> Bool
isSuffix input1 input2 with (snocList input1)
  isSuffix [] 输入2 | 空=真
  isSuffix (xs ++ [x]) 输入2 | (Snoc xsrec) = ?isSuffix_rhs_2
```

3 定义 接下来,匹配第二个输入,再次使用snocList处理

最后一个元素。将光标悬停在?isSuffix_rhs_2 上,按 Ctrl-Alt-W 以
添加嵌套with块:

```
isSuffix : Eq a => List a -> List a -> Bool
isSuffix input1 input2 with (snocList input1)
  isSuffix [] 输入2 | 空=真
  isSuffix (xs ++ [x]) 输入2 | (Snoc xsrec) 和 (snocList input2)
    isSuffix (xs ++ [x]) [] isSuffix (xs ++ [x]) (ys + | (Snoc xsrec) | 空 = ?isSuffix_rhs_2
      + [y]) | (Snoc xsrec) | (Snoc ysrec)
      = ?isSuffix_rhs_3
```

4 Refine 非空列表不能是空列表的后缀,如果列表的最后两个元素相等,您将递归检查列表的其余部分:

```
isSuffix : Eq a => List a -> List a -> Bool
isSuffix input1 input2 with (snocList input1)
  isSuffix [] 输入2 | 空=真
  isSuffix (xs ++ [x]) 输入2 | (Snoc rec) 与 (snocList input2)
```

递归视图:终止和效率

```
isSuffix (xs ++ [x]) [] isSuffix (xs ++ [x]) (ys + [y]) | (Snoc rec) | 空 = 假
+ [y]) | (Snoc rec) | (斯诺克z)
= if x == y then isSuffix xs ys | xsrec | ysrec
否则为假
```

请注意,当您递归调用`isSuffix`时,您会传递两个递归视图参数`xsrec`和`ysrec`,以节省不必要的重新计算它们。

你可以通过在REPL上询问 Idris 来确认这个定义是完整的:

```
*IsSuffix> :总后缀
Main.isSuffix 是总计
```

10.2.4 更多遍历:Data.List.Views

为了帮助您编写总函数,Idris 库提供了许多用于遍历数据结构的视图。 Data.List.Views模块提供了几个,包括您刚刚看到的`SnocList`视图。

例如,清单 10.8 显示了`SplitRec`视图,它允许您递归地遍历一个列表,一次处理一个列表。这类似于您在 10.1.5 节中看到的`SplitList`视图,但在列表的一半上进行了递归遍历。

清单 10.8 Data.List.Views 中的`SplitRec`视图

```
数据 SplitRec :列出一个 -> 键入 where
  SplitRecNil:SplitRec []
  SplitRecOne:SplitRec [x]
  SplitRecPair : (lrec : Lazy (SplitRec lefts)) -> (rrec : Lazy (SplitRec rights)) ->
    SplitRec (左 ++ 右)

总 splitRec : (xs : List a) -> SplitRec xs
```

Lazy 泛型类型Lazy 类型允许

您将计算推迟到需要结果为止。例如,Lazy Int 类型的变量是一个计算,在计算时将产生一个 Int 类型的值。Idris 内置了以下两个函数:

```
延迟 a -> 懒惰
强制:懒惰 a -> a
```

在类型检查时,Idris 将根据需要插入延迟和强制的应用程序。因此,在实践中,您可以将 Lazy 视为说明变量仅在需要其结果时才被评估的注释。您将在第 11 章中更详细地看到 Lazy 的定义。

您可以使用SplitRec将10.1.5 节中的mergeSort重新实现为一个总函数。以下清单显示了我们的起点。

**清单 10.9 使用SplitRec完全实现mergeSort 的起点
(MergeSortView.idr)**

```
导入 Data.List.Views
mergeSort : Ord a => List a -> List a mergeSort input = ?
mergeSort_rhs
```

← 导入 Data.List.Views 以访问 SplitRec

您可以通过以下步骤使用SplitRec视图实现mergeSort：

1 定义首先添加一个with块,表示您想使用SplitRec视图编写函数：

```
mergeSort : Ord a => List a -> List a mergeSort input with
  (splitRec input) mergeSort [] | SplitRecNil = ?mergeSort_rhs_1
    合并排序 [x] | SplitRecOne = ?mergeSort_rhs_2 mergeSort (lefts ++ rights) |
      (SplitRecPair lrec rrec)
      = ?mergeSort_rhs_3
```

2 优化 -输入[]和[x]已排序：

```
mergeSort : Ord a => List a -> List a mergeSort input with
  (splitRec input) mergeSort [] | SplitRecNil = [] 合并排序 [x] |
    SplitRecOne = [x] 合并排序 (左 ++ 右) | (SplitRecPair lrec
      rrec)
      = ?mergeSort_rhs_3
```

3 Refine 对于(lefts ++ rights)的情况,可以递归排序左右,然后合并结果：

```
mergeSort : Ord a => List a -> List a mergeSort input with
  (splitRec input) mergeSort [] | SplitRecNil = [] 合并排序 [x] |
    SplitRecOne = [x] 合并排序 (左 ++ 右) | (SplitRecPair lrec
      rrec) = 合并 (mergeSort lefts | lrec) (mergeSort rights | rrec)
```

该|说,在递归调用中,你想绕过构建视图,因为你已经有适当的左右视图。

你可以确认新的mergeSort定义是total的,并在一些例子上进行测试:

```
*MergeSortView> :total mergeSort Main.mergeSort
是 Total
```

```
*MergeSortView> 合并排序 [3,2,1]
[1, 2, 3] : 列出整数
```

```
*MergeSortView> 合并排序 [5,1,4,3,2,6,8,7,9]
[1, 2, 3, 4, 5, 6, 7, 8, 9] : 列表整数
```

练习



这些练习使用 Idris 库中 Data.List.Views、Data.Vect.Views 和 Data.Nat.Views 模块中定义的视图。对于练习中提到的每个视图,使用:doc 来了解视图及其覆盖功能。

对于这些练习中的每一个,确保伊德里斯认为你的解决方案是全面的。

1 使用 Data.List.Views 中定义的 SnocList 视图实现 equalSuffix 函数。它应该具有以下类型:

```
equalSuffix : Eq a => List a -> List a -> List a
```

它的行为应该是返回两个输入列表的最大相等后缀。这是一个例子:

```
*ex_10_2> equalSuffix [1,2,4,5] [1..5]
[4, 5] : 列出整数

*ex_10_2> equalSuffix [1,2,4,5,6] [1..5] []
: 列出整数

*ex_10_2> equalSuffix [1,2,4,5,6] [1..6]
[4, 5, 6] : 列出整数
```

2 使用 Data.Vect.Views 中定义的 SplitRec 视图为向量实现合并排序。

3 编写一个 toBinary 函数,将 Nat 转换为包含 Nat 二进制表示的字符串。您应该使用 Data.Nat.Views 中定义的 HalfRec 视图。

如果你有一个正确的实现,你应该看到:

```
*ex_10_2> toBinary 42
"101010" :字符串

*ex_10_2> toBinary 94
"1011110" :字符串
```

提示:如果输入是 Z ,可以返回一个空字符串。

4 使用 Data 中定义的 VList 视图编写一个回文函数,该函数返回一个列表在向前和向后遍历时是否相同

.List.Views。

如果您有正确的实现,你应该看到以下内容:

```
*ex_10_2> 回文 (解压 "abccba")
真:布尔

*ex_10_2> 回文 (解压 "abcba")
真:布尔

*ex_10_2> 回文 (解压 "abcb")
假:布尔值
```

提示: VList 视图允许您以线性时间遍历列表,同时处理第一个和最后一个元素并在列表中间递归。

10.3 数据抽象:使用视图隐藏数据结构本章到目前为止你所看到的视图允许你以超出默认模式匹配的方式检查和遍历数据结构,特别关注列表。从某种意义上说,视图允许您描述用于构建模式匹配定义的替代接口:

使用SnocList,您可以看到如何使用[]构造列表,并使用++添加单个元素,而不是使用[]和::。 使用SplitRec,你可以看到一个列表是如何被构造为一个空列表,一个单

吨列表,或一对列表的串联。

也就是说,您可以通过查看视图来了解值是如何构造的,而不是直接查看该值的构造函数。事实上,您通常不需要知道值的数据构造函数是什么才能使用值的视图。

考虑到这一点,视图在实践中的一种用途是隐藏模块中的数据表示,同时仍然允许使用该数据的函数的交互式类型驱动开发,方法是在该数据的视图上拆分大小写。

视图的起源视图的概念是由 Philip Wadler 在 1987 年为 Haskell 提出的,在他的论文“视图:一种模式匹配与数据抽象共存的方式”中。本节中的示例本着 Wadler 论文的精神,其中包含其他几个在实践中使用视图的示例。 Views 作为一种编程习惯用法,使用依赖类型和类似于 Idris 中的with表示法的表示法,后来由 Conor McBride 和 James McKinna 在他们 2004 年的论文“左视图”中提出。

为了结束本章,我们将看看这个想法的实际应用。我们将重温在第 4 章和第 6 章中实现的数据存储示例,隐藏 Idris 模块中的数据表示,并仅导出以下内容:

模式描述,解释存储中数据的形式用于初始化数据存储的函数,空用于向存储添加条目的函数, addToStore 用于遍历存储内容的视图, StoreView 及其覆盖函数, storeView

这些都不需要模块的用户了解存储本身的结构或其中包含的数据结构。

但是,在您实现模块并导出相关定义之前,我们将需要简要讨论 Idris 如何支持模块中的数据抽象。

10.3.1 题外话:Idris 中的模块

除了第 2 章中的一个小示例外,您在本书中编写的程序都独立地包含在一个主模块中。当您编写更大的应用程序时,

但是,您需要一种方法将代码组织成更小的编译单元,并控制从这些单元导出哪些定义。

下面的清单显示了一个小的 Idris 模块,它定义了一个Shape类型并将其导出,以及它的数据构造函数和一个计算形状面积的函数。

清单 10.10 定义形状和面积计算的模块 (Shape.idr)

```
模块形状
公共导出数据 Shape =
Triangle Double Double
|长方形 双 双
|圆形双人间
私人矩形区
域:双->双->双矩形区域宽度高度=宽度*高度
导出区域:
形状 -> 双区域 (三角形底高)= 0.5 *
rectangle_area 底高区域 (矩形长度高度)= rectangle_area 长度高度区域 (圆半径)= pi * 半径 * 半径
```

此模块中定义的每个名称都有一个导出修饰符,用于解释该名称是否对其他模块可见。导出修饰符可以是以下之一:

`private` -根本不导出名称。 导出导出名称和类型,但不导出定义。对于数据类型,这意味着类型构造函数是导出的,但数据构造函数是私有的。

`public export` 名称、类型和完整的定义被导出。对于数据类型,这意味着数据构造函数被导出。对于函数,这意味着函数的定义被导出。

如果函数或数据类型定义上没有导出修饰符,Idris 会将其视为私有的。在前面的示例中,这意味着导入 Shape 的模块可以使用名称 Shape、Triangle、Rectangle、Circle 和 area,但不能使用 rectangle_ 区域。

导出函数定义如果要在类型中使用函数的行为,导出函数的定义及其类型(通过公共导出)很重要。特别是,这对于我们在第 6 章中首次使用的类型同义词和类型级函数很重要。

下一个清单显示了Shape模块的替代版本,它保留了Shape数据类型抽象的详细信息,导出类型而不是其构造函数。

282

第10章视图:扩展模式匹配

清单 10.11 将Shape导出为抽象数据类型 (Shape_abs.idr)

```
模块 Shape_abs
    导出数据
        Shape = Triangle Double Double
            |长方形 双 双
            |圆形 双人间

    导出三角形:
        双 -> 双 -> 形状三角形 = 三角形

    导出矩形:
        双 -> 双 -> 形状矩形 = 矩形

    导出圆:
        -> 形状圆 = 圆
```

导出 Shape 数据类型,但不导出其构造函数

导出用于构建形状的函数,而不是它们的构造函数

在这里,我们导出了用于构建形状的三角形、矩形和圆形函数。与其直接使用数据构造函数,其他模块将需要使用这些函数并且不能对Shape类型进行模式匹配,因为构造函数没有被导出。

使用导出修饰符,您可以实现一个实现数据存储功能的模块,但仅导出用于创建存储、添加项目和遍历存储的函数,而不导出有关存储结构的任何详细信息。

10.3.2 数据存储,再访

为了说明视图在数据抽象中的作用,我们将创建一个实现数据存储的模块,导出用于构建存储的函数。我们还将实现一个用于检查和遍历商店内容的视图。

以下清单显示了DataStore.idr模块。这是一个轻微的变化
在第 6 章中实现的DataStore记录上。

清单 10.12 数据存储,带有模式 (DataStore.idr)

```
模块数据存储
    导入数据.Vect
    中缀 5 .+。
    公共导出数据架构 =
        SString | 整数 | (.+.) 架构 架构

    public export
        SchemaType : Schema -> Type SchemaType
        SString = String SchemaType SInt = Int
        SchemaType (x .+. y) = (SchemaType x,
                                SchemaType y)
```

导出 Schema 及其所有数据构造函数

Schema 类型是在第 6 章中为数据存储实现定义的,以及 SchemaType,它将 Schema 转换为 Idris 类型。

导出 SchemaType 及其定义

```
导出记录
DataStore (schema : Schema) where
  构造函数 MkData
  尺寸:自然
  项目.Vect 大小 (SchemaType 模式)
```

←

←

↓

导出 DataStore, 但不导出其
构造函数或其任何字段
DataStore 记录由数据模式参数化。

这里不是将模式存储为记录中的字段,而是通过数据的模式参数化记录,因为您不打算允许更新模式:

```
导出记录
DataStore (schema : Schema) where 构造函数 MkData
  尺寸:自然
  项目.Vect 大小 (SchemaType 模式)
```

参数化记录声明的语法类似于接口声明的语法,参数及其类型列在记录名称之后。

此声明产生具有以下类型的DataStore类型构造函数:

数据存储:模式 -> 类型

它还产生了将商店的大小(大小)和商店中的条目(项目)投影到记录之外的功能。函数有以下类型:

```
大小:数据存储模式-> Nat
items : (rec : DataStore schema) -> Vect (size rec) (SchemaType schema)
```

因为记录具有导出修饰符export, DataStore数据类型对其他模块可见,但大小和项目投影函数不可见。

清单 10.13 展示了三个函数,其他模块可以使用它们来创建一个具有特定模式(空)的新的空存储,或者向存储添加一个新条目(addTo Store)。这些函数中的每一个都具有导出修饰符export,这意味着其他模块可以看到它们的名称和类型,但无法访问它们的定义。

清单 10.13 访问存储的函数 (DataStore.idr)

```
导出为空:
DataStore 架构为空 = MkData 0 []
```

```
导出
addToStore : (value : SchemaType schema) -> (store : DataStore schema) ->
  数据存储架构
addToStore 值 (MkData 项目)= MkData (价值 :: 物品)
```

为了能够有效地使用这个模块,您还需要遍历商店中的条目。您可以使用empty构建商店的内容来创建新商店,并使用 addToStore添加新条目。因此,能够方便地使用

这些作为模式来匹配商店的内容。当你在 store 上匹配时,你需要处理以下两种情况: 匹配没有内容的 store 的空 case 匹配第一个条目由 value 给出的 store 的 addToStore 值存储 case, 以及店内剩余物品由店家给

为了匹配这些情况,您可以编写 DataStore 的视图。

10.3.3 用视图遍历商店的内容

清单 10.14 展示了一个 StoreView 视图及其覆盖函数 storeView。它们允许您通过查看存储的构造方式 (使用 empty 或 addToStore) 来遍历存储的内容。

清单 10.14 遍历存储中条目的视图 (DataStore.idr)

```
公共导出数据 StoreView:
DataStore 架构 -> 嵌入 where
  SNil:StoreView 为空
  SAdd : (rec : StoreView store) -> StoreView (addToStore value store)

storeViewHelp : (items : Vect size (SchemaType schema)) ->
  StoreView (MkData 大小项) storeViewHelp [] = SNil
  storeViewHelp (val :: xs) = SAdd (storeViewHelp xs)

storeView : (store : DataStore schema) -> StoreView store storeView (MkData size items) =
  storeViewHelp items
```

使用视图时要匹配 StoreView 的构造函数, 所以导出数据构造函数。

没有导出注释, 因此 Idris 认为 storeViewHelp 是私有的。

导出
导出覆盖函数, 以便其他模块可以构建视图

StoreView 视图使您可以通过模式匹配访问商店的内容, 但隐藏其内部表示。要使用 store 并遍历其内容, 您无需了解有关内部表示的任何信息。

为了说明这一点, 让我们设置一些测试数据并编写一些函数来检查它。
下一个清单定义了一个商店并用一些测试数据填充它, 将行星名称映射到首次访问该行星的太空探测器的名称和访问年份。 1

清单 10.15 填充了一些测试数据的数据存储 (TestStore.idr)

```
导入数据存储

testStore : DataStore (SString .+ SString .+ SInt) testStore = addToStore ( Mercury ,
  Mariner 10 , 1974 ) (addToStore ( Venus , Venera , 1961 )
```

¹ 然而, 我们不会在这里讨论冥王星是否是一颗行星。

```
(addToStore ("天王星", "航海者 2", 1986) (addToStore ("冥王星", "新视野", 2015)
空的)) )
```

应用程序运算符 \$ 当表达式嵌套很深时,就像在 testStore 定义中一样,很难跟踪括号。 \$ 运算符是一个中缀运算符,它将函数应用于参数,您可以使用它来减少括号的需要。

使用它,您可以编写以下内容:

```
testStore = addToStore ( Mercury , Mariner 10 , 1974) $
    addToStore ( Venus , Venera , 1961) $ addToStore
    ( Uranus , Voyager 2 , 1986) $ addToStore ( Pluto , New
    Horizons , 2015) $
空的
```

因此,写入 \$ 相当于将表达式的其余部分放在括号中。例如,写 `fx $ yz` 完全等同于写 `fx(yz)`。

以下清单显示了数据存储的基本遍历,返回存储中的条目列表。

清单 10.16 将商店的内容转换为条目列表的函数 (TestStore.idr)

```
listItems : DataStore schema -> List (SchemaType schema) listItems input with (storeView
input) listItems empty | SNil = [] listItems (addToStore 值存储) | (SAdd rec) = value :: listItems
store |记录
```

该 rec 绕过了递归调用中视图的构建,因为 rec 已经是商店其余部分的视图。

如果您使用测试数据调用 `showItems`,您将看到以下结果:

```
*TestStore> listItems testStore
[ ( "水星", "水手 10" ,1974) ,
( "金星", "金星" ,1961) ,
( "天王星", "航海者 2" ,1986 年) ,
( "冥王星", "新视野" ,2015) ]:列表 (字符串,字符串,整数)
```

更有趣的是,您可能想要编写遍历数据存储并过滤掉某些条目的函数。例如,假设您想要获取 20 世纪太空探测器首次访问的行星的列表。您可以通过编写以下函数来做到这一点:

```
filterKeys : (test : SchemaType val_schema -> Bool) -> DataStore (SString .+ val_schema)
-> List String
```

您可以将 `(SString .+ val_schema)` 形式的模式视为提供键值对,其中键是字符串,而 `val_schema` 描述值的形式。

然后, filterKeys将一个函数应用于该对中的值,如果它返回True,它会将键添加到String列表中。这可以找到探测器在2000年之前访问过的行星:

```
*TestStore> filterKeys (\x => snd x < 2000) testStore
[ Mercury , Venus , Uranus ]: 列表字符串
```

您可以通过以下步骤使用StoreView实现filterKeys:

1类型,定义 - 从类型和骨架定义开始:

```
filterKeys : (test : SchemaType val_schema -> Bool) -> DataStore (SString .+ val_schema) -> List
          String
filterKeys 测试输入 = ?filterKeys_rhs_1
```

2定义 您将通过使用StoreView遍历存储来定义函数,因此您可以使用with构造来构建视图,并对其进行大小写拆分:

```
filterKeys : (test : SchemaType val_schema -> Bool) -> DataStore (SString .+ val_schema) -> List
          String
filterKeys 测试输入 (storeView 输入)
filterKeys 测试空 | SNil = ?filterKeys_rhs_1 filterKeys 测试 (addToStore 值存储) | (SAdd
rec) = ?filterKeys_rhs_2
```

3 Refine 如果存储为空,则没有可应用测试的值,因此返回空列表:

```
filterKeys : (test : SchemaType val_schema -> Bool) -> DataStore (SString .+ val_schema) -> List
          String
filterKeys 测试输入 (storeView 输入) filterKeys 测试为空 | SNil = [] filterKeys 测
试 (addToStore 值存储) | (SAdd rec) = ?filterKeys_rhs_2
```

4 Refine 否则,由于数据存储的架构,条目本身必须是键值对:

```
filterKeys : (test : SchemaType val_schema -> Bool) -> DataStore (SString .+ val_schema) -> List
          String
filterKeys 测试输入 (storeView 输入)
filterKeys 测试空 | SNil = [] filterKeys 测试 (addToStore (key,
value) 存储) | (添加推荐)
= ?filterKeys_rhs_2
```

您将对值应用测试。如果结果为True,您将保留密钥并递归构建列表的其余部分。如果结果为False,您将省略键和构建列表的其余部分:

```
filterKeys : (test : SchemaType val_schema -> Bool) -> DataStore (SString .+ val_schema) -> List
          String
filterKeys 测试输入 (storeView 输入) filterKeys 测试为空 | SNil = [] filterKeys 测
试 (addToStore (key, value) 存储) | (添加推荐)
= 如果测试值
```

```
then key :: filterKeys 测试存储 |记录
else filterKeys 测试商店 |记录
```

您可以使用一些测试过滤器尝试此功能:

```
*TestStore> filterKeys (\x => fst x ==  Voyager 2 ) testStore
[ “天王星” ]:列表字符串

*TestStore> filterKeys (\x => snd x > 2000) testStore
[ “冥王星” ]:列表字符串

*TestStore> filterKeys (\x => snd x < 2000) testStore
[ Mercury , Venus , Uranus ]:列表字符串
```

对于showItems和filterKeys,您编写了一个遍历数据存储内容的函数,而无需了解

商店。在每种情况下,您都使用视图来解构数据,而不是直接解构数据。如果您要更改内部表示

DataStore模块,以及对应的storeView的实现,
showItems和filterKeys的实现将保持不变。

练习



1编写一个getValues函数,该函数返回DataStore中所有值的列表。它应该
有以下类型:

```
getValues : 数据存储 (SString .+ val_schema) ->
    列表 (SchemaType val_schema)
```

您可以通过编写一个函数来设置数据存储来测试您的定义:

```
testStore : 数据存储 (SString .+ SInt)
testStore = addToStore( First ,1) $
    addToStore ( 第二 ,2) $
    空的
```

如果您已正确实现getValues,您应该会看到以下内容:

```
*ex_10_3> 获取值 testStore
[1, 2] : 列表整数
```

2定义一个允许其他模块检查抽象Shape数据类型的视图

清单 10.11。您应该能够使用它来完成以下定义:

```
区域:形状 -> 双
area s with (shapeView s)
    面积 (三角形底高) | STriangle = ?area_rhs_1
    面积 (矩形宽度高度) | SRectangle = ?area_rhs_2
    面积 (圆半径) | SCircle = ?area_rhs_3
```

如果您已正确实施此操作,您应该会看到以下内容:

```
*ex_10_3> 区域 (三角形 3 4)
6.0:双倍

*ex_10_3> 区域 (第 10 圈)
314.1592653589793:双
```

10.4 总结

视图是一种依赖类型,它描述了另一种数据类型的可能形式。视图利用依赖模式匹配来允许您扩展可以使用的模式形式。

覆盖函数构建了一个值的视图。按照惯例,它的名字是以小写字母开头的视图名称。

with构造允许您直接使用视图,而无需定义中间函数。

您可以使用视图来定义数据结构的替代遍历,例如

提取列表的最后一个元素而不是第一个元素。

递归视图通过编写与视图模式匹配的递归函数,帮助您编写保证终止的函数。**Idris**在Data.List.Views库中为List的替代遍历提供了几个视图。Vect、Nat和String存在类似的库。您可以隐藏模块中的数据结构,同时仍然支持使用该数据进行交互式类型驱动编程,方法是导出用于遍历数据的视图

结构。

第 3 部分

伊德里斯和现实世界

在 第 2 部分,您获得了交互式开发程序的经验,以类型为指导,您了解了 Idris 的所有核心功能。现在,是时候将您学到的知识应用到一些更实际的例子中了。

首先,在第 11 章中,您将学习编写程序来处理潜在的无限结构,例如流。您已经了解了

写总函数,但在第 11 章你会看到, totality 是关于更多比终止。如果一个函数产生了可能无限结果的一部分,那么它也是完全的,这意味着您可以编写交互式系统,例如服务器和永远运行的读取-评估-打印循环,但它们仍然是完全的。

第 12-14 章讨论状态。现实世界的程序通常需要处理以某种方式使用全局状态,您将看到如何表示状态以及如何以您可以保证程序的方式描述状态的属性准确地遵循协议。如果您正在实施一个重要的系统安全属性,例如银行的 ATM,您可以使用类型驱动开发来确保满足这些属性。

最后,第 15 章提供了一个类型驱动开发的扩展示例,展示了如何实现一个用于并发编程的库类型。您将首先编写一个简单的类型来捕获特定的并发编程问题,然后逐渐对其进行细化以捕获并发程序的更多重要属性。

Machine Translated by Google

11 流和进程： 处理无限数据

本章涵盖

生成和处理数据流

区分终止和生产性总功能

使用无限定义整个交互过程

流

到目前为止,我们在本书中编写的函数都在批处理模式下工作,处理所有输入,然后返回输出。在上一章,我们

还花了一些时间讨论为什么终止很重要,你学会了如何使用视图来帮助您编写保证终止的程序。

但是输入数据并不总是成批到达,你经常想写不终止的程序,无限期地运行。例如,可以方便地将输入数据考虑到交互式程序(如按键、鼠标

运动等)作为连续的数据流,一次处理一个元素时间,导致输出数据流。实际上,许多程序实际上是,流处理器:

一个 read-eval-print 循环 ,例如 Idris 环境 ,处理一个潜在的无限的用户命令流 ,给出响应的输出流。

Web 服务器处理可能无限的HTTP请求流 ,给出一个输出要通过网络发送的响应流。

实时游戏处理来自一个可能无限的命令流控制器 ,给出视听动作的输出流。

此外 ,即使您正在编写不与外部数据源或设备交互的纯函数 ,流也允许您编写可重用的程序组件

通过将数据的生产与数据的消费分开。例如 ,假设您正在编写一个函数来确定数字的平方根。你可以做

这是通过产生一个无限接近解的近似值列表 ,然后编写一个单独的函数来使用该列表 ,找到所需范围内的第一个近似值。

生产和消费数据本章的一个共同主题是消费 (或处理)数据的程序与使用数据的程序之间的区别。

产生数据。到目前为止 ,您在本书中看到的所有函数都是数据的消费者 ,在上一章中 ,我们研究了使用视图来帮助我们编写保证在使用数据时终止的函数。

但是 ,当您编写终止函数时 ,使用数据只是故事的一部分 :生成无限流的函数永远不会毕竟终止。如您所见 ,Idris 检查生成的函数流保证是高效的 ,因此任何消耗的函数流生成器的输出总是有要处理的数据。

我们在实践中编写的程序通常有一个终止组件 ,处理和响应用户输入 ,以及一个非终止组件 ,它是一个

重复调用终止组件的无限循环。在这一章当中 ,您将看到如何编写管理这种区别的程序 ,包括生产和消耗潜在的无限数据。我们将从最常见的无限之一开始结构 ,流 ,然后看看如何定义描述无限执行的交互式函数的总函数。

11.1 Streams:生成和处理无限列表

流是无限的值序列 ,一次可以处理一个值。在本节 ,您将看到如何编写产生无限数据序列的函数 ,根据需要 ,以及如何编写消耗有限部分数据的函数作为流。

作为第一个示例 ,为了说明流背后的想法 ,我们将看看如何生成以及如何将它们处理为执行一个无限的数字序列 , 0、1、2、3、4 ,需要标记列表中 ... ,的元素 ,如图 11.1 所示。

Streams:生成和处理无限列表

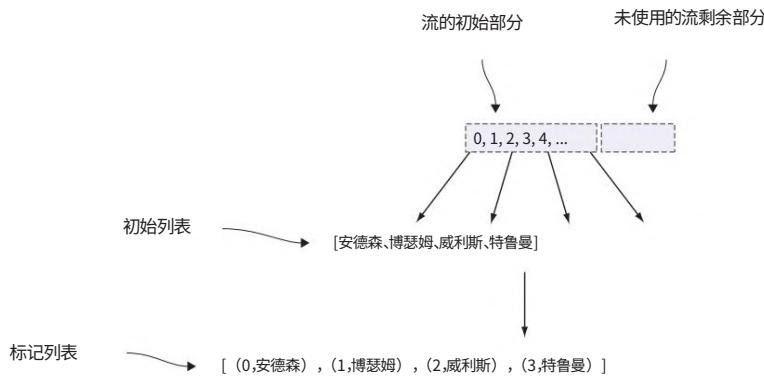


图 11.1 通过从无限的数字流中获取元素来标记列表的元素。流包含无限数量的元素，但您只需要在有限列表中标记元素即可。

您可能会在对某些数据进行排序后使用此类函数，例如，将显式索引附加到数据。正如您将看到的，您可以使用流将无限标签列表的生成与特定输入列表所需的标签的消耗清晰地分开。

除了向您展示如何定义潜在的无限数据类型外，我还将在介绍 Prelude 提供的 Stream 数据类型，我们将简要介绍 Streams 上的一些函数。最后，我们将看一个使用随机数流来实现算术游戏的更大示例。

11.1.1 标记列表中的元素

假设你想编写一个函数，用一个整数来标记 List 的每个元素，指示它在列表中的位置，如图 11.1 所示。也就是说，像这样：

标签 : 列表 a -> 列表 (整数, a)

在某些示例上运行此函数应给出以下结果：

```
*Streams> label [ a , b , c ] [(0, a ), (1, b ), (2, c )] : List (Integer, Char)
*Streams> 标签 [ “Anderson” 、“Botham” 、“Willis” 、“Trueman” ]
[ (0, “安德森” ) ,
  (1, “博瑟姆” ) ,
  (2, “威利斯” ) ,
  (3, Trueman ) ] : List (Integer, String)
```

下面的清单显示了一种通过编写帮助函数来编写标签的方法，labelFrom 将标签作为列表的第一个元素，然后标记列表的其余部分，并增加标签。

清单 11.1 用整数标记列表的每个元素 (Label.idr)

```
labelFrom : Integer -> List a -> List (Integer, a) labelFrom [] = [] labelFrom lbl
(val :: vals) = (lbl, val) :: labelFrom (lbl + 1) vals
```

将标签初始化为 0

标签:列表 a -> 列表 (整数 a) 标签 = labelFrom 0

标记列表的第一个元素,然后递归
标记尾部

这按要求工作,但labelFrom的定义结合了两个组件:标记每个元素,并生成标签本身。另一种编写标签的方法是允许您分别重用这两个组件。您可以编写两个函数:

countFrom 生成一个无限的数字流,从一个向上计数
给定的起点。

labelWith 获取无限的标签流,并将每个标签与有限列表中的对应元素配对。因此,它只消耗尽可能多的无限流来标记列表中的元素。

尝试编写countFrom的一种自然方式可能是从给定的起点生成一个整数列表:

```
countFrom : 整数 -> 列表整数 countFrom n = n :: countFrom
(n + 1)
```

然而,当Idris运行已编译的程序时,它会在评估函数本身之前完全评估函数的参数。因此,不幸的是,如果您尝试将countFrom的结果传递给需要List的函数,该函数将永远不会运行,因为countFrom的结果将永远不会被完全评估。如果你问Idris这个countFrom的定义是否是总的,它会告诉你有问题:

```
*StreamFail> :total countFrom Main.countFrom
由于递归路径,可能不是总数:
Main.countFrom,Main.countFrom
```

您可以看到countFrom永远不会终止,因为它对每个输入进行递归调用,但是要编写labelWith您只需要countFrom结果的有限部分。因此,关于countFrom,您真正需要了解的不是它总是终止,而是它总是会产生您需要的任意数量的数字。

也就是说,您需要知道它是高效的,并且可以保证生成无限长的数字序列。

正如您将在下一节中看到的那样,您可以使用类型来区分那些计算保证终止的表达式和那些保证计算保证不断产生新值的表达式,将数据结构的参数标记为可能无限。

11.1.2 产生一个无限的数字列表

生成无限的数字列表并仅使用列表的有限部分

您可以使用新的数据类型`Inf`来标记潜在的无限结构的部分。

您很快就会看到有关`Inf`如何工作的更多详细信息。不过,首先让我们看一下使用`Inf`的无限列表的数据类型。

清单 11.2 无限列表的数据类型 (InfList.idr)

没有`Nil`构造函数,所以列表没有尽头。

→ 数据 `InfList : 类型 -> 类型在哪里`
`(::) : (value : elem) -> Inf (InfList elem) -> InfList elem`

`Inf`泛型类型标记
`InfList`元素参数
 可能是无限的。

%name InfList xs, ys, zs ← 交互式编辑的名称提示。

`InfList`类似于`List`泛型类型,但有两个显着差异:

没有`Nil`构造函数,只有`(::)`构造函数,所以没有办法结束名单。

递归参数被包装在一个新的数据类型`Inf`中,它标记了参数`ment`作为潜在的无限。

要操纵潜在的无限计算,您可以使用延迟和强制功能。清单 11.3 给出了`Delay`和`Force`的类型。这个想法是你可以使用`Delay`和`Force`可以精确控制何时评估子表达式,这样您就只计算特定函数所需的无限列表的有限部分。

清单 11.3 Inf 抽象数据类型,用于延迟潜在的无限计算

`Inf`是一种潜在无限计算的通用类型。

→ `Inf : 类型 -> 类型`

延迟: `(值:ty) -> Inf ty`
 力: `(计算:Inf ty) -> ty`

延迟是一个函数,它声明它的参数应该只在它的结果被强制时才被评估。

Force 是一个返回延迟计算结果的函数。

以下清单显示了如何定义`countFrom`,生成一个无限列表给定起始值的整数。

清单 11.4 将`countFrom`定义为无限列表 (InfList.idr)

```
countFrom : 整数 -> InfList 整数
countFrom x = x :: 延迟 (countFrom (x + 1))
```

延迟意味着只有在使用`Force`明确请求时才会计算列表的其余部分。

如果您尝试在REPL中评估countFrom 0以生成从 0 向上计数的无限列表 ,您将看到延迟的效果：

```
*Inflist> countFrom 0
0 :: 延迟 (countFrom 1) : Inflist 整数
```

您可以看到 Idris 求值器未对Delay的参数进行求值。评估器特别对待Force和Delay :它只会在 Force明确请求时评估延迟的参数。结果 ,尽管在每个输入上都对countFrom进行了递归调用 ,但REPL的评估仍然终止。伊德里斯甚至同意这是全部：

```
*Inflist> :total countFrom
Main.countFrom 是 Total
```

术语 :递归和修正、数据和协数据你可能会听到 Idris 程序员将诸如countFrom之类的函数称为核心递归而不是递归 ,将无限列表称为协数据而不是数据。数据和余数据之间的区别在于数据是有限的并且旨在被消费 ,而余数据可能是无限的并且旨在被生产。

递归通过获取数据并将其分解为基本情况来运行 ,而核心递归通过从基本情况开始并构建余数据来运行。

伊德里斯认为countFrom是完全的 ,这似乎令人惊讶 ,因为它产生了一个无限的结构。因此 ,在我们继续讨论如何使用无限列表之前 ,有必要更详细地研究一下函数是什么意思是完全的。

11.1.3 题外话 :函数为全是什么意思 ?

如果一个函数是完全的 ,它永远不会因为缺少案例而崩溃 (也就是说 ,所有类型良好的输入都被覆盖) ,并且它总是在有限的时间内返回一个类型良好的结果。

您在前几章中编写的函数都将有限数据作为输入 ,因此只要它们对所有输入都终止 ,它们就是总数。但是现在您已经看到了Inf类型 ,您可以编写产生无限数据的函数 ,并且这些函数不会终止 !因此 ,我们需要完善我们对函数是什么意思的理解。

产生无限数据的函数可以用作终止函数的组件 ,只要它们总是根据请求产生一条新数据。在countFrom的情况下 ,它总是会在进行延迟递归调用之前产生一个新的Integer 。

图 11.2 说明了countFrom 的结构。对countFrom的延迟递归调用是(:)的一个参数 ,这意味着在进行递归调用之前 , countFrom将始终产生无限列表的至少一个元素。因此 ,任何使用countFrom结果的函数都将始终有数据可以使用。



图 11.2 产生无限结构的值。延迟意味着 Idris 只会在 Force 需要时对 countFrom 进行递归调用。

定义的总功能

总函数是对所有类型正确的输入执行以下操作之一的函数：

以良好类型的结果终止

在有限时间内产生一个类型良好的无限结果的非空有限前缀

我们可以将总功能描述为终止或生产。停止问题是难以确定特定程序是否终止，并且，

感谢 Alan Turing, 我们知道一般情况下不可能编写一个程序

解决了停机问题。换句话说,伊德里斯无法确定其中之一是否

条件适用于所有总函数。相反,它通过分析函数的语法来进行保守的近似。

如果存在覆盖所有类型良好的模式,则 Idris 认为函数是完全的
输入,它可以确定以下条件之一成立：

当有一个递归调用 (或一系列相互递归调用) 时,有
向基本情况收敛的递减论点。

当有递归调用作为延迟的参数时,延迟调用将
对于所有输入,在评估后始终是数据构造函数 (或嵌套数据构造函数序列) 的参数。

我们在前一章讨论了这些条件中的第一个。第二个条件允许我们在终止函数中使用类似
`countFrom` 的函数。为了显示
更进一步,了解如何使用生成的无限列表会很有帮助。举个例子,
让我们编写一个函数,该函数使用 `InfList` 的有限长前缀。

11.1.4 处理无限列表

生成 `InfList` 的函数是总的,前提是它保证保持
在需要数据时生成数据。您可以通过编写一个
从无限列表的前缀计算有限列表的程序：

```
getPrefix : (count : Nat) -> InfList ty -> List ty
```

`getPrefix`返回一个由无限列表中的第一个计数项组成的列表。只要它需要更多元素，它就通过递归地从无限列表中获取下一个元素来工作。您可以通过以下步骤对其进行定义：

1 定义 首先，对`count`参数进行大小写拆分：

```
getPrefix : (count : Nat) -> InfList a -> List a
getPrefix Z xs = ?getPrefix_rhs_1
getPrefix (S k) xs = ?getPrefix_rhs_2
```

2 Refine 如果你从无限列表中取零个元素，返回一个空的列表：

```
getPrefix : (count : Nat) -> InfList a -> List a
getPrefix Z xs = []
getPrefix (S k) xs = ?getPrefix_rhs_2
```

3 定义 如果你有多个元素，你将在无限列表上进行大小写拆分，然后将无限列表中的第一个值添加为结果。

```
getPrefix : (count : Nat) -> InfList a -> List a
getPrefix Z xs = []
getPrefix (S k) xs = value :: xs
value :: ?getPrefix_rhs_1
```

4 类型，细化如果您查看孔的类型`?getPrefix_rhs_1`，您会看到以下：

```
a : 类型 k : Nat
值：一个
xs : Inf (InfList a)
-----
getPrefix_rhs_1 : 列出一个
```

从`xs`的类型可以看出它是一个尚未计算的无限列表，因为它是一个封装在`Inf`中的无限列表。要完成定义，您可以强制计算`xs`并递归获取其前缀：

```
getPrefix : (count : Nat) -> InfList a -> List a
getPrefix Z xs = []
getPrefix (S k) xs = value :: getPrefix k (Force xs)
```

根据 Idris 的说法，得到的定义是完全的：

*`InfList`:total `getPrefix Main.getPrefix` 是 Total

即使其中一个输入可能是无限的，`getPrefix`也只会评估从无限列表中检索计数元素所需的数量。因为`count`是一个有限数，所以只要保证`InfList`继续产生新元素，`getPrefix`就会一直终止。

在实践中，您可以省略对延迟和强制的调用，让 Idris 在需要的地方插入它们。如果在类型检查期间，Idris 在

需要ty类型的值,它将添加对Force的隐式调用。同样,如果它在需要Inf ty时遇到ty,它会添加对Delay的隐式调用。以下清单显示了如何使用隐式Force和Delay定义countFrom和getPrefix。

清单 11.5 获取无限列表的有限部分,隐含Force和Delay (InfList.idr)

```
countFrom : 整数 -> InfList 整数 countFrom x = x :: countFrom (x + 1)
getPrefix : Nat -> InfList a -> 列出一个 getPrefix Z x = [] getPrefix (S k)
(x :: xs) = x :: getPrefix k xs
```

← Idris 类型检查器隐式插入递归调用所需的延迟。

← Idris 类型检查器隐式插入 xs 所需的 Force。

因此,您可以将Inf视为类型上的注释,标记数据结构中可能无限的部分,并让Idris类型检查器管理何时必须延迟或强制计算的详细信息。

现在您已经了解了如何分离数据的产生,使用countFrom生成无限的数字列表,从数据的消费中分离出来,使用getPrefix之类的函数,我们可以重新审视标签的定义。与其使用我们自己的InfList数据类型和countFrom,我们将使用Prelude中为此目的定义的数据类型: Stream。

11.1.5 流数据类型

清单 11.6 显示了Prelude中Stream的定义。它与您在上一节中看到的InfList的定义具有相同的结构。此外,Prelude还提供了一些用于构建和操作Streams的有用函数,其中一些也显示在此列表中。

清单 11.6 Prelude 中定义的Stream数据类型和一些函数

```
数据流:类型 -> 类型在哪里
(:) : (value : elem) -> Inf (Stream elem) -> Stream elem
```

生成特定元素的无限列表

```
重复: elem -> Stream elem take: (n : Nat) -> (xs :
Stream elem) -> List elem iterate: (f: elem -> elem) -> (x : elem) -> Stream elem
```

通过重复应用函数生成流

从流的开头获取特定数量的元素

您可以在REPL中看到函数重复、执行和迭代。例如, repeat生成一个元素的无限序列,延迟到特别请求:

伊德里斯>重复94
94 :: 延迟 (重复 94) 流整数

与InfList上的getPrefix一样，take采用特定长度的Stream的前缀：

```
Idris> 采取10 (重复94)
[94, 94, 94, 94, 94, 94, 94, 94, 94] :列表整数
```

iterate 重复应用一个函数，生成一个结果流。例如，从0开始并重复应用(+1)会导致一系列递增的整数，例如countFrom：

```
Idris> 取 10 (迭代 (+1) 0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] :列表整数
```

流生成的语法糖

Idris 提供了一种用于生成数字流的简洁语法，类似于列表的语法：

```
Idris> 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] :列表整数
```

语法[1..]生成从 1 向上计数的Stream。这适用于任何可数数字类型，如下例所示：

```
Idris> (List Int) 取 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] :列表整数
```

您还可以更改增量：

```
Idris> (List Int) (取10 [1..3])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19] :列表整数
```

将所有这些放在一起，下面的清单显示了如何使用iterate定义标签以生成无限的整数标签序列，以及一个labelWith函数，该函数消耗足够的无限标签序列来将标签附加到List 的每个元素。

清单 11.7 使用Stream (Streams.idr) 标记List 的每个元素

使用泛型类型变量 labelType，因为 labelWith 从不检查标签本身

→ labelWith : 流 labelType -> List a -> List (labelType, a) labelWith [] = [] labelWith (lbl :: lbls) (val :: vals) = (lbl,

val) :: labelWith lbls vals

label : List a -> List (Integer, a) label = labelWith (iterate (+1) 0)

用标签流中的第一个元素标记列表的第一个元素

没有留下任何标签，所以返回一个空列表

在此定义中,您将生成标签和为列表的每个元素分配标签这两个组件分开。

您甚至可以为labelWith 提供更通用的类型,采用Stream labelType而不是Stream Integer,并允许按您可以为其生成Stream 的任何类型进行标记。例如,循环函数生成一个重复给定的Stream

序列:

```
*Streams> take 10 $ 循环 [ a , b , c ][ a , b ,
c , a , b , c , a ]:列表字符串
```

使用循环生成流,您可以使用循环的标签序列标记列表的每个元素,根据需要重复多次以标记整个列表:

```
*Streams> labelWith (cycle[ a , b , c ]) [1..5] [( a ,1),( b ,2),( c ,3),( a ,4),
( b ,5]):列表 (字符串,整数)
```

11.1.6 使用随机数流的算术测验

您可以在需要数据源但事先不知道需要生成多少数据的情况下使用Stream 。例如,您可以编写一个交互式算术测验,该测验采用问题的数字源。以下清单显示了您可以如何执行此操作。

清单 11.8 算术测验,为问题取一个数字流(Arith.idr)

```
quiz : Stream Int -> (score : Nat) -> IO () quiz (num1 :: num2 :: nums) score = do putStrLn
( Score so far: putStrLn (show num1 ++ answer <- getLine if cast answer == 数字 1 * 数
字 2
+
"?" )
++ 显示分数)++显示num2 ++
"?" )
```

从 Ints 的无限源中获取两个数
字

正确答案;继续增加分数

然后做 putStrLn “正确!” 测验数字 (分数 + 1)

```
else do putStrLn( 错了,答案是
测验分数
错误的答案;显示正确答案并以相同分数继续
```

++ 显示 (num1 * num2))

quiz函数获取Ints的无限源和到目前为止的分数,然后返回一个显示分数和问题的IO操作,读取用户的答案,然后重复。这些问题直接来自输入流,因此您可以尝试使用一系列递增的数字:

```
*Arith> :exec 测验 (迭代 (+1) 0) 0
目前得分:0
0 * 1? 0
正确的!
目前得分:1
2 * 3? 6
正确的!
```

```
目前得分:2
4 * 5? 20
正确的!
目前得分:3
```

ABORTING EXECUTION重复循环程序的执行将继续,直到某些外部事件导致程序退出。您可以通过按 Ctrl-C 在 REPL 处中止执行。

到目前为止,这并不是特别有趣,因为您事先知道问题是什么将会。相反,您可以编写一个生成伪随机流的函数从初始种子生成的整数。以下清单显示了一种生成使用线性同余生成器的随机数字流。

清单 11.9 从种子 (Arith.idr) 生成伪随机数流

```
随机数:Int -> Stream Int
随机种子 = 让种子 = 1664525 * 种子 + 1013904223 in
          (种子 `shiftR` 2) :: 随机种子
将当前种子乘以一个常数,然后加上另一个
shiftR 按位移动一个整数
就在给定数量的地方
```

伪随机数生成 清单 11.9 中的 `randoms` 函数 使用线性同余生成器从初始种子生成看起来随机但可预测的数字流。这是最古老的伪随机数生成技术之一。它适合我们的目的

例如,但它不适合高质量随机性的情况
由于生成数字的分布和可预测性,因此需要加密应用程序。

您可以尝试使用随机生成的数字流运行测验:

```
*Arith> :exec quiz (randoms 12345) 0
目前得分:0
1095649041 * -2129715532?不知道
错了,答案是-765659660
目前得分:0
```

早些时候,这些问题太容易预测了,但现在它们可能有点太难了
对于我们大多数人来说!此外,在前面的示例中,结果甚至超出了 Int 的范围,因此报告的答案不正确。相反,下一个清单显示了一种处理随机结果的方法,以便它们在合理的范围内

算术测验的界限。

清单 11.10 为测验生成合适的输入(Arith.idr)

```
导入 Data.Primitives.Views
arithInputs : Int -> Stream Int
arithInputs 种子 = 地图绑定 (随机种子)
您需要将其导入 Divides
视图
```

在哪里
 绑定 :Int -> Int
 将 num 与 (除以 num 12) 绑定
 绑定 ((12 * div) + rem) | (DivBy prf) = rem + 1

通过除以 12 时的被除数和余数来描述 num 匹配

使用 Divides 视图进行安全除法

您可以通过除以 12 并检查余数来将任意 Int 减少到 1 到 12 之间的值。除法不一定安全，因为除以 0 在 Int 上未定义，因此您可以使用 Int 的视图来解释数字为 0 或由乘法加余数组成：

```
data Divides : Int -> (d : Int) -> Type where
  DivByZero : Int.Divides x 0
  DivBy : (prf : rem >= 0 && rem < d = True) ->
```

```
Int.Divides ((d * div) + rem) d
```

划分：(val:Int) -> (d:Int) -> 划分 val d

请注意，在 DivBy 的情况下，您还可以通过使用第 8 章介绍的相等类型= 来证明余数的值在 0 和除数之间。

您可以使用 arithInputs 作为测验的随机输入源，并确保所有问题都使用 1 到 12 之间的数字。下面是一个示例：

```
*Arith> :exec 测验 (arithInputs 12345) 0
目前得分: 0
2 * 1? 2
正确的!
目前得分: 1
6 * 2? 18
错了,答案是12
目前得分: 1
11 * 10? 110
正确的!
目前得分: 2
```

您已成功使用随机 ints 源作为测验的输入，并且由于随机数（以及因此 arithInputs）产生无限的数字序列，只要您需要，您就可以生成新数字。

然而，还有一个问题，即测验本身并不完整：

```
*Arith> :total quiz Main.quiz 可能不是
全部,因为递归路径:Main.quiz, Main.quiz
```

这不应该让您感到惊讶，因为测验的定义中没有任何内容允许它终止。相反，与 countFrom、randoms 和 arithInputs 一样，它读取用户输入并不断产生无限序列的 IO 操作。

在实践中，交互式程序通常有一个外循环，你可以运行它无限期地，调用特定的命令，你想终止每个命令你可以发出下一个命令。编写运行的交互式程序的方法因此，无限期地是区分描述描述的交互式程序的类型从描述可能无限的动作序列（主循环）的交互式程序类型中终止动作序列（我们的主循环中的命令）

本身）。我们将在下一节进一步探讨这一点。

练习



1 编写一个 `every_other` 函数，从输入流的每一秒元素中产生一个新流。

如果您正确地实现了这一点，您应该会看到以下内容：

```
*ex_11_1> 取 10 (every_other [1..])
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20] ·列表整数
```

2 为 `InfList` 编写 `Functor` 的实现（如第 7.3.1 节所述）。

如果你正确地实现了这个，你应该会看到以下内容，使用 `count` 11.1.4 节中定义的 `From` 和 `getPrefix`：

```
*ex_11_2> getPrefix 10 (map (*2) (countFrom 1))
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20] ·列表整数
```

3 定义代表硬币面的人脸数据类型：正面或反面。然后，
定义这个：

```
coinFlips : (count : Nat) -> Stream Int -> List Face
```

这应该使用流作为源返回一系列计数硬币翻转
随机性。如果你正确地实现了这个，你应该会看到类似
以下：

```
*ex_11_3> coinFlips 6 (随机数 12345)
[尾巴, 头, 尾巴, 尾巴, 头, 尾巴] ·列表表面
```

提示：这将有助于定义一个函数 `getFace : Int -> Face`。

4 您可以定义一个函数来计算 `Double` 的平方根，如下所示： 1 生成一系列更接近平方根的近似值。

2 取第一个近似值，当平方时，它在所需的范围内
原来的号码。

编写一个生成一系列近似值的函数：

```
square_root_approx : (number : Double) -> (approx : Double) -> Stream Double
```

在这里，您正在寻找数字的平方根，从一个近似值开始，
大约您可以使用以下公式生成下一个近似值：

下一个 = (大约 + (数字 / 大约)) / 2

如果您正确地实现了这一点,您应该会看到以下内容:

```
*ex_11_1> 取 3 (square_root_approx 10 10)
[10.0, 5.5, 3.659090909090909] : 双列表
```

```
*ex_11_1> 取 3 (square_root_approx 100 25)
[25.0, 14.5, 10.698275862068964] : 列表双
```

3编写一个函数,找到一个平方根的第一个近似值

期望的界限,或在最大迭代次数内:

```
square_root_bound : (max : Nat) -> (number : Double) -> (bound : Double) -> (approxs : Stream Double) -> Double
```

如果max为零,这应该返回approxs的第一个元素。否则,它应该返回第一个元素(我们称之为val),其中val x val和number之间的差异小于界限。

如果你正确地实现了这个,你应该能够定义square_root
如下,应该是总数:

```
square_root : (number : Double) -> Double
square_root number =
square_root_bound 100 number 0.0000000001
          (square_root_approx 数)
```

您可以使用以下值对其进行测试:

```
*ex_11_1> square_root 6
2.449489742783178:双
```

```
*ex_11_1> square_root 2500 50.0:双倍
```

```
*ex_11_1> square_root 2501
50.009999000199954:双倍
```

11.2 无限进程:编写交互式总程序

当你编写一个函数来生成一个Stream时,你给出了Stream的前缀并递归地生成余数。这与测验类似,因为您让初始IO操作运行,然后递归生成剩余的IO操作。因此,您可以将交互式程序视为产生可能无限序列的交互式操作的程序。

在第5章中,您使用IO泛型类型编写了交互程序,其中IO ty是终止交互动作的类型,给出类型ty的结果。在本节中,您将了解如何通过定义InfIO类型来表示无限的IO操作序列,来编写非终止但高效(因此也是完全)的交互式程序。

因为InfIO描述了无限的动作序列,所以函数必须是高效的,因此您可以确定生成的程序会继续产生新的IO动作以供执行,同时继续永远运行。但是InfIO仅描述了交互操作的序列,因此您还需要编写一个函数来执行这些操作。

总的来说,我们将采用以下方法来编写交互式总程序:

1定义一个类型InflO来描述无限的交互动作序列。

2使用InflO编写非终止但高效的函数。

3定义一个将InflO程序转换为IO动作的运行函数。

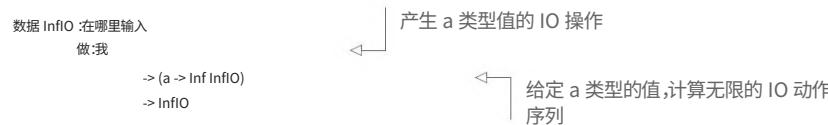
你的第一次运行本身不会是全部的。然后你会看到如何细化

定义使 even run是总的,使用数据类型来描述执行应该持续多长时间。不过,首先,您需要了解如何通过定义InflO 来描述无限过程。

11.2.1 描述无限过程

以下清单显示了如何定义InflO类型。它类似于Stream,除了也就是说,在交互式程序中,您可能希望第一个动作产生的值影响计算的其余部分。

清单 11.11 无限交互进程 (InflO.idr)



Do有两个论据:

要执行的IO操作的描述

无限动作序列的剩余部分。因为它的类型是 $a \rightarrow \text{Inf}$

InflO,它可以使用第一个动作产生的结果来计算剩余的动作。

通过使用Inf将序列的其余部分标记为无限,您就是在告诉 Idris

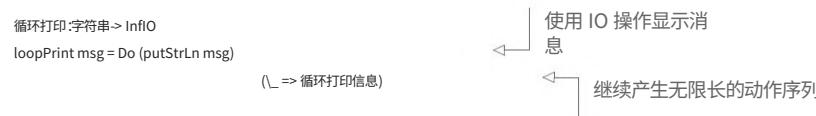
您希望返回InflO类型值的函数具有生产力。其他

换句话说,与Stream 上的函数一样,延迟递归调用会产生InflO

必须是构造函数的参数。下面的清单显示了它是如何在一个

重复显示消息的递归程序。

清单 11.12 重复显示消息的无限进程 (InflO.idr)



loopPrint的递归调用是构造函数Do和loopPrint的参数

保证产生一个构造函数 (Do)作为其结果的有限前缀。这满足

11.1.3 节中生产性总函数的定义,所以 Idris 很高兴

loopPrint是总计:

```
*InflO> :total loopPrint Main.loopPrint
是 Total
```

回顾第 5 章, IO 是一种描述交互动作的通用类型, 它将由运行时系统执行。如果您尝试在 REPL 上评估 loopPrint, 您将看到将执行的第一个 IO 操作的描述以及无限操作序列的延迟剩余部分:

```
*InflO> loopPrint 你好
做 (io_bind (prim_write "Hello!\n") (\__bindx => io_return ()) ) (\underline{underscore} => Delay
    (loopPrint "Hello" )) : InflO
```

与 IO 一样, 要使其在实践中有用, 您还需要能够执行无限的操作序列。

11.2.2 执行无限进程

在第 5 章中, 您了解了 Idris 运行时系统如何执行 IO ty 类型的程序, 其中 ty 是交互式计算产生的值的类型。因此, 为了执行 InflO 类型的值, 您需要首先将其转换为 IO ()。

这是执行此操作的一种方法。

清单 11.13 将 InflO 类型的表达式转换为可执行的 IO 操作 (InflO.idr)

```
run : InflO -> IO () run (Do action
cont) = do res <- action run (cont res)
           | 要执行的第一个动作是 Do 的
           | 第一个参数。
           | 通过将操作的结果 res 传递给计算
           | 的其余部分 cont 来继续执行
```

使用 run, 您可以将重复打印消息的函数转换为 IO 操作, 并使用 :exec 执行它:

```
*InflO> :exec run (loopPrint "on and on and on..." ) on and on and on...
```

```
不断...          | 无限期地继续
不断...          |
```

因为它会无限期地运行, 至少在您按 Ctrl-C 中止它之前, 您可能不会惊讶地发现 Idris 不认为 run 是全部的:

```
*InflO> :total run Main.run 由
于递归路径可能不是总的 :Main.run, Main.run
```

你知道 loopPrint 将继续产生新的 IO 动作来执行, 因为它是总的。这很有价值, 因为继续执行 IO 操作的程序将继续取得明显的进展 (至少假设这些操作产生一些输出, 我们将在 11.3.2 节中进一步考虑)。如果 run 也是全部的, 那就太好了, 这样您至少会知道所有可能的 IO 操作都已处理并且

运行本身的实现不会导致任何意外的非终止。

这似乎是不可能的:拥有一个完整的、非终止函数的唯一方法是使用`Inf`类型,而`IO`是一种不使用`Inf`的终止动作。而且,确实,如果您希望函数在运行时无限期地执行,您至少需要某种方法来逃避所有函数。但是,您可以尝试尽可能安静地逃跑。

为了实现这一点,我们将首先制作一个`run`的终止版本,它将作为参数是它愿意执行的操作数量的上限。

11.2.3 将无限进程作为总函数执行

之前,在 11.1.4 节中,您编写了一个`getPrefix`函数来检索无限列表的有限部分:

```
getPrefix : (count : Nat) -> InfList a -> List a
```

您可以将`count`参数视为允许您继续处理无限列表的“燃料”。一旦燃料耗尽,您将无法再处理列表中的任何内容。你可以对`run`做类似的事情,给它一个额外的参数来代表它将运行的迭代次数。

下面的清单定义了一个`Fuel`数据类型,并给出了一个新的、总计的定义只要有关燃料,就会执行动作。

**清单 11.14 将`InfIO`类型的表达式转换为可执行的`IO`操作
运行有限时间 (`InfIO.idr`)**

燃料 定义了时间 的长度 进程可以运行。

数据燃料 = 干燥 |更多燃料

tank : Nat -> Fuel tank Z = Dry
tank (S k) = More (tank k)

产生一定量的燃料。为 Fuel 定义一个新类型,而不是使用 Nat 你很快就会明白为什么。

run : Fuel -> InfIO -> IO () run (More fuel) (Do cf) =
do res <- c run fuel (f res)

run Dry p = putStrLn 燃料耗尽

消耗一滴燃料并继续执行

没有更多的燃料;放弃执行

现在,运行是总的:

```
*InfIO> :total run Main.run 是
Total
```

不幸的是,您仍然有一个问题,因为您现在需要明确指定程序允许执行的最大操作数,因此您实际上不再有无限期运行的进程!例如:

```
*InfIO> :exec run (tank 5) (loopPrint vroom )
弗鲁姆
弗鲁姆
```

```

弗鲁姆
弗鲁姆
弗鲁姆
弗鲁姆
没油了

```

确保run是完全的很有价值,因为它保证run本身的实现不会成为任何意外未终止的原因。但是,如果您仍希望程序无限期运行,则需要找到一种无限期生成燃料的方法。您可以通过使用惰性数据类型来实现这一点。

11.2.4 使用惰性类型生成无限结构

如果你有办法生成无限燃料,你可以无限期地运行交互式程序。清单 11.15 展示了如何永远使用一个非全函数来做到这一点。您还需要更改Fuel的定义,使其在类型中明确表示您仅在需要时生成Fuel。

清单 11.15 生成无限燃料 (InflO.idr)

```

数据燃料 = 干燥 |更多 (惰性的燃料)
永远:燃料
永远=永远更多

```

稍后解释的 Lazy 意味着仅在需要时才计算参数的值。

根据需要产生燃料

FOREVER AND NONTERMINATION永远是非完全的,因为它(故意)引入了非终止。幸运的是,这是您能够永远执行程序所需的唯一非全部功能。

Lazy的目的是控制 Idris 何时计算表达式。正如名字Lazy所暗示的那样,Idris 不会计算Lazy ty类型的表达式,直到Force 明确请求它,它返回一个ty 类型的值。 Prelude 定义Lazy的方式类似于您在 11.1.2 节中定义的Inf :

```

懒惰:类型 -> 类型
延迟: (值:ty) -> Lazy ty
强制: (计算:惰性 ty) -> ty

```

此外,与Inf一样, Idris 会隐式插入对Delay和Force的调用。事实上, Inf和Lazy非常相似,以至于它们在内部使用相同的底层类型实现,如下面的清单所示。 Inf和Lazy在实践中的唯一区别是整体检查器处理它们的方式,如边栏中所述。

清单 11.16 Inf和Lazy 的内部定义

```

数据延迟原因 = 无限 |惰性值
数据延迟:DelayReason -> Type -> Type where
    延迟 : (val : ty) -> 延迟原因 ty

```

一个计算被延迟,要么是因为它可能是无限的,要么是因为它要在以后进行评估。



使用 `Inf` 和 `Lazy` 进行总体检查在运行时, `Inf` 和

`Lazy` 的行为方式相同。它们之间的主要区别在于整体检查器处理它们的方式。Idris 通过寻找收敛于基本情况的参数来检测终止,因此它需要知道构造函数的参数是否比整个构造函数表达式更小(即更接近基本情况) :

如果参数具有`Lazy ty` 类型,对于某些类型`ty`,它被认为小于构造函数表达式。

如果参数的类型为`Inf ty`,对于某些类型`ty`,它不会被认为小于构造函数表达式,因为它可能会无限地继续扩展。相反,Idris 将检查整个表达式是否有效,如第 11.1.3 节所述。

如果您将`Inf`用于`Fuel`,而不是`Lazy`,则`run`将不再是总的,因为参数`fuel`不会被认为小于表达式`More fuel`。

您已经实现了三个函数: `loopPrint`,它是交互式程序;运行,在给定燃料的情况下执行交互式程序;并且永远,它提供了无限量的燃料。总结一下:

`loopPrint`是一个总函数,因为它不断产生IO动作 `indefi`
晚上。

`run`是一个总函数,因为它会消耗IO动作,将它们执行为
只要有可用的燃料。

`forever`是一个非全(或部分)函数,因为它永远不会终止并且
不会在`Inf`类型内产生任何数据。

通过编写一个`run`版本,只要有燃料就可以处理数据,Idris 可以保证`run`是完全的,在运行时会消耗燃料。
你仍然有一个“逃生舱”,允许你以永远功能的形式无限期地运行交互式程序。然而,永远是唯一不完整
的功能。

您仍然可以改进`loopPrint`本身的定义。当我们在第 5 章中编写交互式程序时,我们使用`do`表示法来帮助使交互式
程序更具可读性,但使用`InfIO`却无法做到这一点。但是,您可以扩展`do`表示法以支持您自己的数据类型,例如`InfIO`。

11.2.5 为 InfiO 扩展 do 表达法

正如你在第 5 章中看到的，do 表达法转换为($>=$)运算符的应用，如图 11.3 所示。



图 11.3 在排序操作时使用 $>=$ 运算符将 do 表达法转换为表达式

你已经在第 5 章看到了 IO 的这种转换，在第 6 章看到了 Maybe 的转换，以及在第 7 章看到了 Monad 接口的一般实现。事实上，这种转换是纯语法的，所以你可以定义你自己的实现($>=$)为您自己的类型使用 do 表达法。以下清单显示了如何为 InfiO 定义 do 表达法。

清单 11.17 为无限的动作序列定义 do 表达法

```

(>=): IO a -> (a -> InfiO) -> InfiO
(>=) = 做
  ↗ 直接使用 InfiO 中的 Do 定义 (>=)
  运算符

loopPrint : String -> InfiO
loopPrint msg = do
  putStrLn msg
  ↗ Idris 将此 do 块转换为 (>=) 运算
  符的应用程序。
  ↗ 循环打印消息
  
```

Idris 将 do 块转换为($>=$)的应用程序，并通过查看所需的类型来决定使用哪个版本的($>=$)。在这里，因为整个表达式所需的类型是 InfiO，所以它使用($>=$)的版本来生成 InfiO 类型的值。

InfiO 类型允许您描述无限运行的交互式程序，并且通过定义($>=$)运算符，您可以像使用 IO 的程序一样编写这些程序，前提是最终操作是调用该类型的函数信息组织。

现在您已经看到可以使用 do 编写高效的交互式程序符号，我们可以从第 11.1.6 节重新审视算术测验。

11.2.6 总算术测验

结束本节时，我们将更新算术测验，使其成为一个全函数，您将了解如何将其合并到一个完整的 Idris 程序中。清单 11.18 显示了我们的起点，设置 InfiO 类型和运行函数，正如您在本节前面所见。您需要 Data.Primitives.Views 来生成随机数流。您还将为 time 函数导入 System 模块，您将使用它来帮助初始化随机数流。

清单 11.18 设置 InfiO (ArithTotal.idr)

```

导入 Data.Primitives.Views 导入系统
  ↗ 您将需要这个时间函数，您将使用它来播种随机流。
  
```

%默认总计

数据 InflO :在哪里输入
执行:IO a -> (a -> Infl InflO) -> InflO

(>>=): IO a -> (a -> Infl InflO) -> InflO
(>>=) = 做

该编译器指令意味着,除非另有说明,否则所有函数都应该是全部的。

%default 总计指令

Idris 支持许多编译器指令，这些指令可以更改有关语言的某些细节。在清单 11.18 中，`%default total` 指令意味着如果有任何函数不能保证是总的，Idris 将报告错误。

您可以使用partial关键字覆盖单个函数。例如，forever不是全部：

部分永远：燃料

永远=永远更多

在您的程序中使用`%default total`是个好主意,因为如果 Idris 无法确定函数是终止的还是生产的,这可能表明函数的定义存在问题。此外,明确标记哪些函数是部分函数意味着如果存在非终止问题,或者由于缺少输入而导致程序崩溃,您已经将可能导致问题的函数数量降至最低。

清单 11.19 展示了下一步，使用 InflO 实现测验。因为 InflO 是无限序列的 IO 操作，所以您可以像以前一样编写测验，最后一步是递归调用。事实上，这个定义和前面的定义是一样的；只有类型发生了变化。

清单 11.19 定义一个总测验函数 (ArithTotal.idr)

```
quiz : Stream Int -> (score : Nat) -> InflO quiz (num1 :: num2 :: nums) score = do
putStrLn ( Score so far:  putStrLn (show num1 ++ answer <- getLine if (cast
    answer == 数字 1 * 数字 2)
        ++显示分数++显示num2 +
        .
        + "? " ) )
```

最后一步是对测验的递归调用。这些调用是富有效果的，因为它们每个都遵循一系列 IO 操作。

++ 显示 (num1 * num2)

因为您使用的是`%default total`注释,所以您可以确定测验是总的。quiz有两次递归调用,Idris可以确定每个调用都保证以一系列IO动作作为前缀,因此quiz保证无限期地继续产生IO动作。

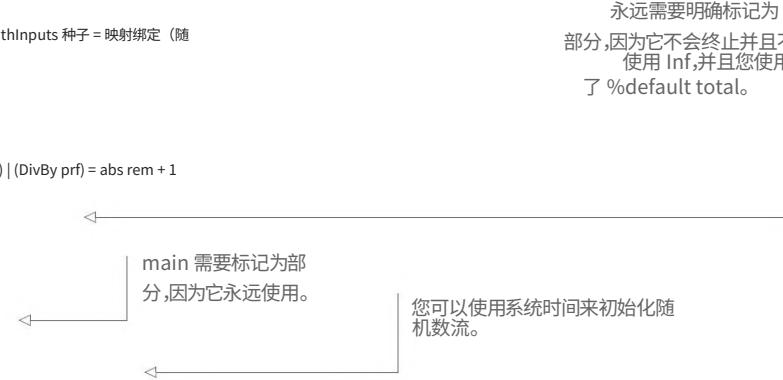
最后一步是编写一个main函数,调用run来执行测验整数流。以下是实现的其余部分。

清单 11.20 使用main (ArithTotal.idr) 完成实现

```
随机数:Int -> Stream Int
随机种子 = 让种子 = 1664525 * 种子 + 1013904223 in
    (种子 `shiftR` 2) :: 随机种子

arithInputs : Int -> Stream Int arithInputs 种子 = 映射绑定 (随
机种子)
在哪里
    绑定:Int -> Int
    将 x 与 (除以 x 12)
        绑定 ((12 * div) + rem) | (DivBy prf) = abs rem + 1

部分永远:燃料
永远=永远更多
部分主要:IO ()
主要=做种子<-时间
永远运行 (quiz (arithInputs (fromInteger seed)) 0)
```



您已经使用了 11.1.6 节中定义的randoms和arithInputs来生成ints 流。通过使用系统时间来初始化流,每次运行程序时都会得到不同的问题。

在整个实现中,唯一不完全的函数是forever和main,后者只是因为它需要forever来生成无限量的运行燃料。因为您使用了`%default total`注释,所以您需要将它们显式标记为部分。这意味着您可以确定程序未终止的唯一可能原因是您故意说程序应该永远运行。

除了永远,您知道各个组件将是以下之一:

高效 例如测验继续产生IO动作以进行解释,随机数在需要时继续产生新的随机数终止 例如单个 IO命令和生成

种子中的下一个随机数

这种区别对于编写诸如服务器和REPL 之类的真实程序很有用,您希望这些程序无限期地运行,同时确保程序执行的每个单独的操作都会终止。但是,您通常需要更多的灵活性。例如,目前您无法彻底退出测验。我们将在下一节中回到这一点。

锻炼



Prelude 中定义的repl函数并不完整,因为它是一个无限循环的IO动作。使用InflO实现新版本的repl：

如果你正确地实现了这个, totalREPL应该是总的,你应该能够按如下方式测试它:

11.3 带终止的交互式程序

使用Inf,您可以明确控制何时生成数据或何时使用数据。因此,您可以在始终终止的程序和永远继续运行的程序之间进行选择。但是,要编写完整的应用程序,您需要更多的控制权。毕竟,虽然您希望服务器无限期地运行,但您希望能够在需要时干净地关闭它。

到目前为止,您使用Inf定义的类型只有一个构造函数,因此它们要求您生成无限序列。相反,您可以在单一数据类型中混合无限和有限组件,这意味着您可以描述可以无限期运行但也允许终止的进程。在本节中,您将看到如何优化InflO类型以支持干净终止的进程。此外,您将看到如何在类型中引入更高的精度,并专门为控制台I/O 定义一种进程类型。

11.3.1 精炼 InflO:引入终止

使用InflO,您可以编写保证不断产生IO操作并无限期运行的全部交互式程序。以下清单显示了您在上一节中看到的表单的一个小示例。该程序反复询问用户的姓名并显示问候语。

清单 11.21 反复问候用户,永远运行 (Greet.idr)

通常,当您编写交互式程序时,您会希望为用户退出。不幸的是,用户退出greet的唯一方法是按 Ctrl-C。
没有办法在InfIO中编写退出任何其他方式的函数!

幸运的是,您可以通过对InfIO 稍作改动来解决这个问题。 Inf类型标记一个潜在无限的值,而不是保证该值是无限的,并且您可以为潜在的无限类型引入额外的数据构造函数。你可以定义一个新的RunIO类型,显示在下一个清单中,它添加了一个Quit构造函数来描述退出的程序,产生一个值。

清单 11.22 RunIO类型,用 附加退出命令 (RunIO.idr)

RunIO 由交互式进程产生的值类型参数化,如果它终止的话。

→ 数据 RunIO : 类型 -> 类型在哪里
退出:a -> RunIO a
执行:IO a -> (a -> Inf (RunIO b)) -> RunIO b

(>>=) : IO a -> (a -> Inf (RunIO b)) -> RunIO b
(>>=) = 做

退出,产生价值
由单个 IO 操作组成的进程,
后跟一个可能无限的进程

实现 (>>=) 以支持 RunIO 程序的
do 表示法

使用RunIO,您可以编写一个版本的greet,如以下清单所示,
当用户给出一个空输入时退出。

清单 11.23 反复问候用户,在空输入时退出 (RunIO.idr)

```
问候:RunIO ()  
greet = do putStrLn "输入你的名字:name <- getLine  
    如果名字 == ""  
        然后做 putStrLn "Bye bye!"  
        退出 ()  
    否则做 putStrLn ("你好"打招呼  
++名称)  
    输入为空,因此显示一条消息  
    并退出。  
    在这里,greet 正在终止。  
有输入,所以向用户打招呼并继续。递归调用遵循 IO 操作,因此 greet 是高效的。
```

根据输入, greet要么是终止的,要么是生产的。整体性 checker 接受greet作为总,因为它满足第 11.1.3 节的定义
对于所有类型良好的输入,一个总函数要么终止,要么是有效的。

在执行greet 之前,您需要编写一个新版本的run ,它将 RunIO中的程序转换为一系列IO操作,以便运行时系统执行。以前,运行只会在燃料用完时终止,但现在有

终止的两个可能原因:

- 燃料耗尽,和以前一样
- 如果正在执行的进程调用Quit ,则干净地退出

交互式程序和无限类型使用像RunIO这样的潜在无限数据

类型并不是 Idris 独有的。 Peter Hancock 和 Anton Setzer 在他们 2004 年的论文 “依赖类型理论中的交互式程序和弱最终代数”中描述了类似的想法,这是在他们早期描述具有依赖类型的交互式程序的工作之后。

Nils Anders Danielsson 在他 2010 年的论文 “Total Parser Combinators”中描述了在潜在无限结构类型中使用的泛型Inf遵循 Agda 编程语言中使用的类似想法。

您可以在类型中区分这些结果：

运行 : Fuel -> RunIO a -> IO (Maybe a)

运行的可能结果对应于两种可能的终止原因：

如果用完Fuel,则返回Nothing。 如果run执行形式为Quit 值
的动作,则返回Just 值。

下面的清单给出了RunIO的新定义。

清单 11.24 将RunIO ty类型的表达式转换为IO ty (RunIO.idr)类型的可执行操作

run : Fuel -> RunIO a -> IO (Maybe a) run fuel (Quit value) = pure
(Just value) run (More fuel) (Do cf) = do res <- c run fuel (f res)

由于调用 Quit 命令而终止,因此它返回
Just 值

运行 Dry p = pure Nothing

由于燃料耗尽而终止,因此它返回
Nothing

最后,您可以编写一个执行greet并丢弃结果的主程序：

部分主要:IO ()
主要~永远运行问候纯 ()

因为run现在创建一个IO动作,当用greet调用时会产生一个类型为Maybe ()的值,而main应该创建一个产生一个类型为() 的值的动作,所以您需要通过调用pure() 来完成。当你在REPL 中执行main时,你现在可以通过输入一个空字符串来干净地退出：

```
*RunIO> :exec 主程序
输入您的姓名:埃德温
你好埃德温
输入你的名字:再见!
———— 空字符串
```

使用RunIO,您可以改进InfIO ,使其能够在需要时干净地终止进程。这使您可以更自由地编写交互式程序,
但是

RunIO可以说给你太多自由的另一种方式。具体来说， RunIO描述的过程是一系列任意IO操作,为您提供多种可能性,包括：

- 读写控制台
- 打开和关闭文件
- 打开网络连接
- 删除文件

对于您在本章中编写的程序,您只对第一个感兴趣
这些。其他的不仅没有必要,而且在第三种情况下可能会导致远程安全漏洞,在第四种情况下
可能会出现破坏性错误。

我们在第 1 章中讨论过的类型驱动开发的原则之一,
是我们应该致力于编写尽可能精确地描述值的类型
住那种类型。因此,在结束本章时,我们将看看如何改进
RunIO类型仅描述实现算术测验所需的操作。

11.3.2 特定领域的命令

实现算术测验时只需要两个IO操作:从
并写入控制台。因此,您可以将它们限制为仅执行您需要的操作,而不是允许RunIO中的交互
式程序执行任意操作。

也就是说,您可以阻止您的程序执行任何交互式操作
在您工作的问题域之外。

下一个清单显示了一个精炼的ConsoleIO类型,它描述了只支持从控制台读取和写入的
交互式程序。

清单 11.25 仅支持控制台 I/O 的交互式程序 (ArithCmd.idr)

```
数据命令 :类型 -> 键入位置
    PutStr :字符串 -> 命令 ()
    GetLine :命令字符串

    定义可用的 IO 命令,由命令的返回
    类型参数化。

数据 ConsoleIO :类型 -> 在哪里键入
    退出:a -> ConsoleIO a
    执行:命令 a -> (a -> Inf (ConsoleIO b)) -> ConsoleIO b

    只有在 Command 中定义的 IO 操作
    在 ConsoleIO 程序中是允许的。
    (=>=) : 命令 a -> (a -> Inf (ConsoleIO b)) -> ConsoleIO b
    (=>=) = 做
```

实际上, Command定义了ConsoleIO程序可以使用的交互界面。

您可以将其视为定义交互式程序的功能或权限,
消除任何不必要的动作。

您现在需要细化run的实现以能够执行ConsoleIO
程式。

清单 11.26 执行ConsoleIO程序 (ArithCmd.idr)

```

runCommand : 命令 a -> IO a
runCommand (PutStr x) = putStr x
运行命令 GetLine = getLine

运行: Fuel -> ConsoleIO a -> IO (Maybe a)
运行燃料 (Quit val) = 做纯 (Just val)
运行(更多燃料) (Do cf) = do res <- runCommand c
运行燃料 (f res)
运行 Dry p = pure Nothing

```

使用相应的 IO 操作运行控制台 I/O 命令

使用 runCommand 执行
IO 动作
由命令 c 给出

领域特定语言(DSL)是一种语言

专门针对特定类别的问题。 DSL通常旨在仅提供处理特定问题时所需的操作

域中的专家可以访问该域的符号,同时消除任何冗余操作。

从某种意义上说, ConsoleIO定义了一个用于编写交互式控制台程序的DSL ,因为它将程序员限制为只能执行以下交互操作:
需要并消除不必要的操作,例如文件处理或网络通信。

清单 11.27 展示了如何修改quiz以作为ConsoleIO程序运行。通过查看quiz的类型以及ConsoleIO 和run的定义,您可以保证quiz只会执行控制台I/O操作。没有办法打开或

关闭文件、通过网络通信或执行任何其他类型的交互
手段。

清单 11.27 算术测验,编写为ConsoleIO程序 (ArithCmd.idr)

您将允许玩家退出,因此 quiz 在退出时返回玩
家的分数。

```

测验 : Stream Int -> (score : Nat) -> ConsoleIO Nat
测验 (num1 :: num2 :: nums) 分数
= do PutStr ( 目前得分: ++ show score ++ \n )
    PutStr (显示 num1 ++ 答案 <- GetLine ++ 显示 num2 ++ ? )

```

PutStr 和 GetLine 有效
ConsoleIO 中的命令。

```

if toLower answer == quit then Quit score else
    if (cast answer == num1 * num2)
        然后执行 PutStr Correct\n
        测验数字 (分数 + 1)
    else do PutStr ( 错了,答案是 show (num1 * num2) ++ \n ) ++

```

输入“退出”将退出
测验。

测验分数

要完成实现,您需要实现一个main函数。下面的清单显示了一个新的main实现,它执行测验,然后

显示玩家进入退出后的最终得分。

清单 11.28 执行测验并显示最终分数的主函数
(ArithCmd.idr)

这不会发生,因为您永远使用,但最好是覆盖
尽管如此,所有可能的运行结果。

显示测验结果产生的分
数

您仍然可以稍微细化测验的定义。随着功能越来越大,这很好。
练习将它们分解为更小的功能,每个功能都有明确定义的角色。
例如,您可以在此处提取用于报告答案正确或错误的功能,如下面的清单所示。这些函数本身必须是

有成效的 (通过调用测验来完成,就像他们在这里所做的那样)或者如果测验要终止
保持总量。

清单 11.29 将quiz的组件提取到单独的函数中 (ArithCmd.idr)

在相互块中 (见第 3 章) ,
定义可以相互引用。
你需要相互因为正确
和
错误都调用测验,反之亦然。

测验类型检查成功后,您可以通过查看类型并检查整体性来对其行为做出若干保证:

除了putStr和getLine之外,它不会执行任何交互动作。
它要么立即退出,要么执行至少一个交互动作,因为
它是富有成效的。
执行的每个动作都会在有限时间内返回一个结果,因为它是全部的。

11.3.3 使用 do 表示法对命令进行排序

在实现quiz时,您使用了两种类型来构造函数: Command和安慰:

命令描述了单个命令,这些命令会终止。ConsoleIO描述了终止命令的序列,可能是无穷。

因此,您可以拥有单个有限命令或无限命令序列。但是能够构造复合命令也很有用;也就是说,保证终止的命令序列。例如,您可能想编写一个显示提示的复合命令,然后读取并解析用户输入。下一个清单显示了表示可能的用户输入的类型,以及用于读取和解析输入的函数的骨架定义。

清单 11.30 定义表示用户输入的类型和复合命令 用于读取和解析输入 (ArithCmdDo.idr)

```
数据输入 = 答案整数
|退出命令
readInput : (prompt : String) -> 命令输入 readInput prompt = ?readInput_rhs
```

输入是回答问题的数字或退出命令。

返回类型 Command Input 意味着 readInput 不会无限循环。

要编写此函数,您需要执行以下操作: 1显示提示。

- 2从控制台读取输入。
- 3将输入字符串转换为Input并返回。

由于Command当前仅支持单个命令,因此您需要对其进行扩展以支持命令序列。下一个清单显示了扩展定义,包括两个新的数据构造函数Pure和Bind,以及相应更新的runCommand定义。

清单 11.31 扩展命令以允许命令序列 (ArithCmdDo.idr)

```
什么都不做
的命令,返
回一个值
数据命令 :类型-> 键入位置
  PutStr : 字符串 -> 命令 ()
  GetLine : 命令字符串
    纯 : ty -> 命令 ty
    绑定 :命令 a -> (a -> 命令 b) -> 命令 b
runCommand : 命令 a -> IO a
runCommand (PutStr x) = putStr x runCommand
GetLine = getLine runCommand (Pure val) = pure val
runCommand (Bind cf) = do res <- runCommand c
  运行命令 (f res)
    两个命令的序列,获取第一个命令
    的输出并将其传递给第二个命令
    运行第一个命令,然后运行
    第二个
    第一个的输出
```

您可能还想定义($>=$)以支持排序命令的do表示法,但以下定义并不像您预期的那样工作:

```
(>=) : 命令 a -> (a -> 命令 b) -> 命令 b
(>=) = 绑定
```

如果你尝试这个,Idris 会抱怨($>=$)已经定义了,因为你已经为ConsoleIO 定义了do表示法:

```
ArithCmdDo.idr:22:7:Main.>= 已定义
```

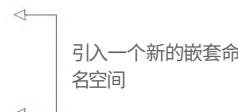
Idris 允许同一函数名的多个定义,只要它们位于单独的命名空间中。例如,您已经在List和Vect 中看到了这一点,其中每个具有名为Nil和 $(::)$ 的构造函数,Idris 根据您使用它们的上下文消除歧义。

命名空间由定义函数的模块给出。命名空间也是分层的,因此您可以在模块中引入更多的命名空间。你在一个模块中可以有多个函数定义 ($>=$) 每个都有新的命名空间。以下清单显示了如何为每个($>=$) 定义新的名称空间。

清单 11.32 在不同的命名空间中创建($>=$)的两个定义(ArithCmdDo.idr)

```
命名空间 CommandDo
(>=) : 命令 a -> (a -> 命令 b) -> 命令 b
(>=) = 绑定
```

```
命名空间 ConsoleDo
(>=) : 命令 a -> (a -> Inf (ConsoleIO b)) -> ConsoleIO b
(>=) = 做
```



如果你在REPL中检查($>=$)的类型,你会看到($>=$)的所有定义它们各自的命名空间,以及它们的类型:

```
*ArithCmdDo> :t (>=)
Main.CommandDo.(>=) : 命令 a ->
                           (a -> 命令 b) -> 命令 b
Main.ConsoleDo.(>=) : 命令 a ->
                           (a -> Inf (ConsoleIO b)) -> ConsoleIO b
Prelude.Monad.(>=) : Monad m => ma -> (a -> mb) -> mb
```

为 Command 定义 Monad 实现

您还可以为Command 定义Monad接口的实现,如在第 7 章中进行了描述。如果可能,这通常是更可取的,因为 Prelude 和基础库定义了几个与Monad实现通用的函数。为此,您还需要定义Functor和

适用。您将在下一个示例中看到如何为类似类型执行此操作章节。

(继续)

但是,您不能为ConsoleIO定义Monad的实现,因为
ConsoleDo.(>=)的类型不适合Monad中(>=)方法的类型
界面。

使用CommandDo.(>=)提供do表示法,可以完成定义
读取输入。

清单 11.33通过序列命令实现readInput (ArithCmdDo.idr)

```
readInput : (prompt : String) -> 命令输入
readInput 提示 = 做 PutStr 提示
    答案 <- GetLine
    如果toLowerCase answer == “退出”
        然后是 Pure QuitCmd
        否则纯 (答案 (接答案))
```

最后,您可以在主测验函数中使用readInput来封装细节
显示提示并解析用户的输入,如最终定义所示。

清单 11.34使用readInput定义quiz作为复合命令来显示 提示并读取用户输入 (ArithCmdDo.idr)

```
测验 : Stream Int -> (score : Nat) -> ConsoleIO Nat
测验 (num1 :: num2 :: nums) 分数
    = do PutStr ( Score so far:  input <- readInput (show ++ 显示分数 ++ \n )
        num1 ++ case input of
                    .                                ++ 显示 num2 ++ ? )
    回答答案 => 如果答案 == num1 * num2
        然后纠正 nums 分数
        否则错误的 nums (num1 * num2) 分数
    QuitCmd => 退出分数
```

您可以对输入进行大小写拆分,因为
readInput 已将其解析为比简单的字符
串更能提供信息的类型。

在这个最终定义中,您可以区分命令的终止序列
(使用命令),以及潜在的非终止控制台I/O程序(使用
控制台IO)。从语法上讲,您在每个函数中都以相同的方式编写函数,但类型告诉
您是否允许该函数无限期运行,或者它是否最终必须运行
终止。

练习

- 1更新测验,以便它跟踪问题的总数,然后返回
正确答案的数量和尝试的问题数量。一个样品
运行可能看起来像这样:

```
*ex_11_3> :执行
目前得分:0/0
9 * 11? 99
正确的!
目前得分:1/1
6 * 9? 42
错了,答案是 54
目前得分:1/2
10 * 22 20
正确的!
目前得分:2/3 7 * 2? 最终得分:2 / 3
```

2从第 11.3.2 节扩展命令类型,使其他支持读取和

写文件。

提示:查看 Prelude 中 readFile 和 writeFile 的类型来决定什么
您的数据构造函数应该具有的类型。

3 使用您的扩展命令类型来实现支持以下命令的交互式“shell”:

```
cat [filename],读取文件并显示其内容
copy [source] [destination],读取源文件并将其内容写入目标文件

exit,退出shell
```

11.4 总结

您可以使用 Inf 生成无限数据来说明结构的哪些部分可能是无限的。一个总函数要么以类型良好的输入终止,
要么产生一个前缀

在有限时间内的良好类型的无限结果。

你可以通过使用有限的数据来处理无限的结构来确定多少
要使用的无限结构。

Prelude 定义了一种 Stream 数据类型,用于构造无限列表。您可以将流程定义为 IO 操作的
无限序列。要执行一个无限过程,你定义一个接受参数的函数

说明该过程应该运行多长时间。

偏函数永远允许进程通过生成

无限量的燃料,使用 Lazy 类型。

通过实现(>>=),您可以将自己的数据类型的 do 表示法扩展为
使程序更易于阅读和编写。

您可以在同一数据类型中混合使用有限和无限结构来定义可能也可能终止的无限进程。

编写带有状态的程序

本章涵盖

使用State类型来描述可变状态为状态管理实现自定义类
型为全局系统状态定义和使用记录

Idris 是一种纯语言,因此变量是不可变的。一旦使用值定义了变量,就无法更新它。这可能表明编写操纵状态的程序是困难的,甚至是不可能的,或者 Idris 程序员通常对状态不感兴趣。在实践中,情况恰恰相反。

在类型驱动的开发中,函数的类型准确地告诉您函数在其允许的输入和输出方面可以做什么。所以,如果你想编写一个操作状态的函数,你可以这样做,但你需要在函数的类型中明确说明它。事实上,我们在前面的章节中已经这样做了:

在第 4 章中,我们使用全局状态实现了一个交互式数据存储。在第 9 章中,我们实现了一个猜词游戏,使用全局状态来保存目标词、猜中的字母以及仍然可用的猜词次数。

在第 11 章中,我们实现了一个算术游戏,使用全局状态来保存用户当前的分数。

在每种情况下,我们通过编写一个递归函数来实现状态整个程序的当前状态作为参数。

几乎所有现实世界的应用程序都需要在某种程度上操纵状态。有时,如前面的示例,状态是全局的,并且在整个应用程序中使用。有时,状态对于算法来说是本地的;例如,图遍历

算法会将其访问过的节点保持在本地状态,以避免更多地访问节点不止一次。在本章中,我们将了解如何在 Idris 中管理可变状态,既适用于算法本地的状态,也适用于表示整个系统状态。

状态管理和依赖类型我们不会使用很多依赖本章中的类型;在状态中使用依赖类型会带来一些复杂性,以及精确描述状态转换系统和协议的机会。我们将在接下来的两章中考虑这些机会,

但我们将从这里开始了解状态的一般运作方式。

以前,我们使用类型来描述交互程序的序列命令,使用第 5 章中的 IO 和第 11 章中的 ConsoleIO。有状态程序可以以同样的方式工作,使用类型来描述有状态程序可以执行的操作履行。我们将从查看 Idris 库中定义的通用 State 类型开始然后是我们如何自己定义像 State 这样的类型。最后,我们将看看我们如何可以用状态构造一个完整的应用程序,从第 11 章。

12.1 使用可变状态

尽管 Idris 是一种纯语言,但使用状态通常很有用。在编写具有复杂数据结构(例如树或图形,以便在您遍历结构时能够读取和写入本地状态。在这个部分,您将看到如何管理可变状态。

比较 IDRIS 和 HASKELL中的状态本节描述状态用于在 Idris 中捕获本地可变状态的类型。如果你熟悉 Haskell,你会发现你可以像 Haskell 一样在 Idris 中使用 State,通过导入 Control.Monad.State。如果您熟悉 State 类型 Haskell,您可以放心地继续阅读第 12.2 节,我将在其中描述 State 的自定义实现。

在上一章中,您编写了一个将标签附加到列表元素的函数,从 Stream 中获取标签。在此重复以供参考。

清单 12.1 使用 Stream (Streams.idr) 标记 List 的每个元素

```
labelWith : 流 labelType -> List a -> List (labelType, a)
标签与磅 [] = []
labelWith (lbl :: lbls) (val :: vals) = (lbl, val) :: labelWith lbls vals
```

对于流中的每个标签,将其与列表的对应元素。

在本节中,您将实现一个类似于标记二叉树的函数。首先,您将看到如何手动实现可变状态,每个函数返回一个存储更新状态和计算结果的对。然后,您将看到如何使用Idris基础库中定义的State类型来封装状态。不过,首先,我将更详细地描述树遍历的示例。

12.1.1 树遍历示例

图 12.1 显示了标记二叉树的函数的结果,我们将在本节中将其用作运行示例。此函数首先标记节点深度,从左到右,因此最深、最左边的节点采用第一个标签,最深、最右边的节点采用最后一个标签。

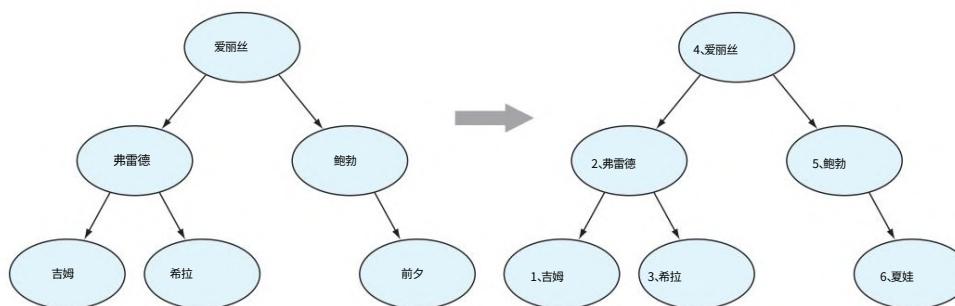


图 12.1 标记树,深度优先。每个节点都标有一个整数。

下面的清单给出了我们将用于标记的二叉树的定义,以及testTree,我们将标记的特定树的表示(来自图 12.1 中的示例)。

清单 12.2 二叉树的定义和示例 (TreeLabel.idr)

具有值、左子树和右子树的树节点	<pre>Tree a = Empty > 节点 (树 a) a (树 a)</pre>	←—— 一个空的二叉树数据
	<pre>testTree : 树字符串 testTree = Node (Node (Node Empty Jim Empty) Fred (节点空 “希拉”空) “爱丽丝” (节点为空 “Bob” (节点为空 “Eve”为空)))</pre>	←—— 示例树的表示
	<pre>展平 :树 a -> 列出 a flatten Empty = [] flatten (Node left val right) = flatten left ++ val :: flatten right</pre>	
	通过遍历树,深度优先,从左到右将树转换为列表	

定义flatten很方便,这样您就可以轻松查看标签的应用顺序:

*TreeLabel> 展平测试树
[“Jim”、“Fred”、“Sheila”、“Alice”、“Bob”、“Eve”]:列表字符串

一旦您编写了一个根据流中的元素标记树中节点的treeLabel函数,您应该能够按如下方式运行它:

```
*TreeLabel> 展平 (treeLabel testTree)
[ (1, "吉姆") ,
  (2, "弗雷德") ,
  (3, "希拉") ,
  (4, "爱丽丝") ,
  (5, "鲍勃") ,
  (6, Eve) ] : 列表 (整数、字符串)
```

当您编写函数来标记列表时,在清单 12.1 中,您在标签流的结构和您正在标记的列表之间建立了直接对应关系。

也就是说,对于列表中的每个(::),您可以将流的第一个元素作为标签,然后递归地标记列表的其余部分。使用树,它有点复杂,因为当你标记左子树时,你事先不知道你需要从流中获取多少元素。图 12.2 说明了在此示例中标记子树。

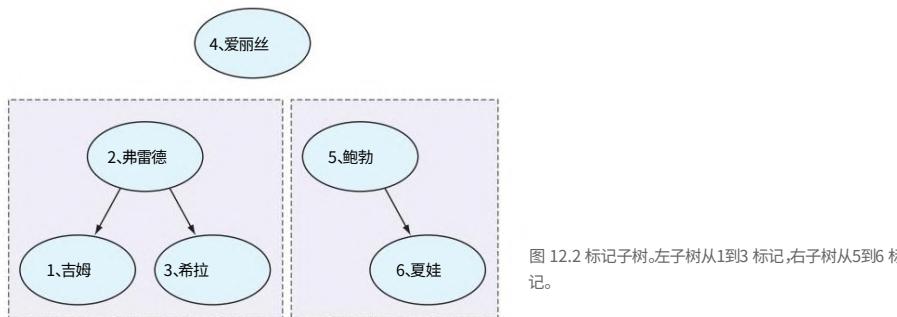


图 12.2 标记子树。左子树从1到3 标记,右子树从5到6 标记。

在标记右子树之前,您需要知道在标记左子树时从流中获取了多少元素。标记函数不仅需要返回标记的树,还需要返回一些关于从何处开始标记树的其余部分的信息。

一个自然的方法可能是标签函数像以前一样将标签流作为输入,并返回一个对,包含

标记树

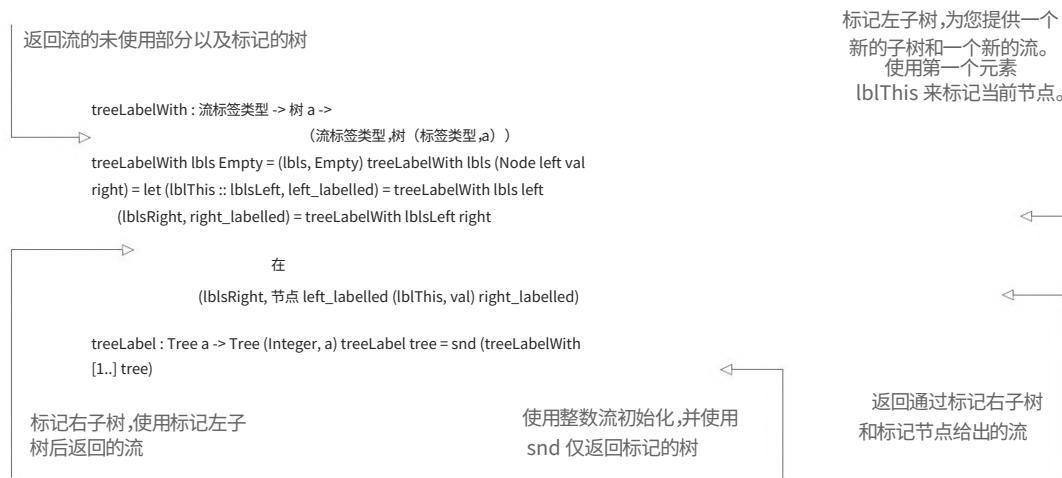
可用于标记树的其余部分的新标签流

我们将从以这种方式实现树标记开始,使用一对来表示操作的结果和标签的状态。然后,您将看到 Idris 库中定义的State类型如何在这种算法中封装状态管理细节。

12.1.2 使用对表示可变状态

清单 12.3 定义了一个辅助函数来用标签流标记树，并使用一对来表示每个子树被标记后流的状态。辅助函数返回标签流中未使用的部分，这样当您标记了一个子树时，您就知道从哪里开始标记下一个子树。

清单 12.3 用标签流标记树(TreeLabel.idr)



如果您尝试为示例树添加标签，您会看到标签按您期望的顺序应用：

```

*TreeLabel> 展平 (treeLabel testTree)
[ (1, "吉姆") ,
  (2, "弗雷德") ,
  (3, "希拉") ,
  (4, "爱丽丝") ,
  (5, "鲍勃") ,
  (6, Eve )]: 列表 (整数,字符串)

```

您可以通过省略调用来检查标签是否也保留了树的结构
变平：

```

*TreeLabel> treeLabel 测试树
节点 (Node (Node Empty (1, Jim ) Empty)
      (2, "弗雷德")
      (节点空 (3, "希拉") 空))
      (4, 《爱丽丝》)
      (节点空
      (5, "鲍勃"))
      (Node Empty (6, Eve ) Empty)): Tree (Integer, String)

```

在treeLabelWith 的当前定义中，您需要跟踪标签流的状态。标记子树不仅为您提供了一个带有附加到节点的标签的树，它还为您提供了一个新的标签流，用于标记树的下一部分。

当您遍历树时,您通过将流作为参数,并返回更新的状态。本质上,该函数使用本地可变状态,状态显式贯穿定义。虽然这有效,这种方法有两个问题:

这很容易出错,因为您需要确保传播正确的状态正确的递归调用。注意left_labelled和right_labelled例如,具有相同的类型,因此很容易使用错误的类型!很难阅读,因为算法的细节隐藏在状态管理的细节。

拥有本地可变状态通常很有用,就像您经常使用的任何概念一样,它是创建一个捕捉概念的抽象是一个好主意。你可以改进treeLabel的定义,通过使用明确捕获状态概念的类型。

12.1.3 State,一种描述有状态操作的类型

在树 LabelWith 的定义中传递标签流的唯一原因是,当您在节点处遇到值时,需要将其与

一个标签值。在命令式语言中,您可以将可变变量传递给树 LabelWith 并在遇到每个节点时对其进行更新。因为 Idris 是一种纯函数式语言,所以您没有可变变量,但基础库确实提供了一个

用于描述有状态操作序列的类型,在 Control.Monad.State 模块。Control.Monad.State 导出以下相关定义。

清单 12.4 状态和相关函数,在 Control.Monad.State 中定义

一种描述状态操作序列的类型,状态类型为 stateType,产生类型为 ty 的结果

运行一系列有状态的操作,产生一对结果和最终状态

状态: (stateType : 类型) -> (ty : 类型) -> 类型
runState : 状态 stateType a -> stateType -> (a, stateType)

写入一个新状态,产生一个 () 类型的结果

获取:状态 stateType stateType
put : stateType -> 状态 stateType ()

读取当前状态,产生 stateType 类型的结果

序列 get 和 put 使用 do 表示法。
有一个 Monad 实现
定义 (>>=) 的状态。

正如 IO ty 的值描述了一系列交互操作,这些操作产生一个 ty 类型的值, State Nat ty 类型的值描述了一系列操作。读取和写入 Nat 类型的可变状态。清单 12.5 给出了一个处理可变状态的函数的小例子。它使用 get 读取状态,然后更新它。使用 put,将 Nat 状态增加给定值。

清单 12.5 将状态增加给定值的有状态函数 (State.idr)

```
导入 Control.Monad.State
增加 : Nat -> State Nat ()
增加 inc = do current <- get put (current + inc)
               ↑———— 状态类型需要
               ↑———— 将状态的值分配给当前
               ↑———— 更新状态的值
```

State Nat ()类型的值是对使用 Nat 类型状态的有状态操作的描述。您可以使用runState通过将操作和初始状态传递给它来执行它。

例如,您可以使用初始状态89执行增加 5 :

```
*State>runState (增加5)89
((), 94) : (), 纳特
```

结果是有状态操作产生的一对值,在本例中为单位值(),在本例中为最终状态94。runState还有两种变体:

evalState 只返回操作序列产生的值:

```
*状态>:t evalState
evalState : 状态 stateType a -> stateType -> a

*State> evalState (增加5) 89 () : ()
```

execState 仅返回操作序列后的最终状态:

```
*状态>:t execState
execState : 状态 stateType a -> stateType -> stateType

*State> execState (增加5) 89
94:纳特
```

状态的一般类型

如果您检查 get 和 put 的类型,您会发现它们使用受约束的泛型类型:

```
*TreeLabelState>:t get : MonadState
stateType m => m stateType

*TreeLabelState>:t put put : MonadState
stateType m => stateType -> m ()
```

这使库作者在定义有状态程序时具有更大的灵活性。 MonadState 接口的详细信息超出了本书的范围,但您可以在 Idris 库文档(<http://idris-lang.org/documentation>) 中阅读更多内容。在本例中,您可以将 m 读作 State stateType。

尽管 State 封装了有状态操作的序列,但在内部它是使用纯函数定义的。本质上,它封装了用于在 treeLabelWith 中传递状态的实现模式。

使用State,您可以重新实现treeLabelWith,隐藏状态管理的内部细节,只在必要时读取和更新标签流。

12.1.4 带状态的树遍历

下一个清单显示了如何通过将标签流保持为状态来定义treeLabelWith ,读取它以获取节点的下一个标签。

清单 12.6 将treeLabelWith 定义为一系列有状态操作 (TreeLabelState.idr)

```
treeLabelWith : Tree a -> State (Stream labelType) (Tree (labelType, a)) treeLabelWith Empty = pure Empty treeLabelWith
(Node left val right) = do left_labelled <- treeLabelWith left (this :: rest) <- get put rest right_labelled <- treeLabelWith
right pure (Node left_labelled (this, val) right_labelled)
```



在这个定义中,可以比之前的定义更清楚地看到标注算法的细节。在这里,您将状态管理的内部细节留给State 的实现。

为了运行此函数并实际执行树标记,您需要提供初始状态。下面的清单定义了一个treeLabel函数,它使用无限的整数流初始化状态,从1 开始向上计数。

清单 12.7 标记树节点的顶级函数,深度优先,从左到右 (TreeLabelState.idr)

```
treeLabel : Tree a -> Tree (Integer, a) treeLabel tree = evalState
(treeLabelWith tree) [1..]
```

evalState 丢弃最终状态,因此它只返回带标签的树。

和以前一样,您可以在REPL 上进行测试。它的行为方式与之前在测试输入上实现treeLabel的方式相同:

```
*TreeLabelState> 展平 (treeLabel testTree)
[ (1, "吉姆") ,
(2, "弗雷德") ,
(3, "希拉") ,
(4, "爱丽丝") ,
(5, "鲍勃") ,
(6, Eve )]: 列表 (整数、字符串)
```

就像您在第 5 章中第一次遇到的IO一样, State通过描述操作序列并分别执行它们,为您提供了一种编写具有副作用的函数 (这里是修改可变状态)的方法:

`IO ty`类型的值是对产生 `ty` 类型结果的一系列交互动作的描述。它由运行时系统通过编译执行，

或通过REPL的`:exec`。

`State`类型的值`stateType ty`是对一系列动作的描述

读取和写入`stateType` 类型的状态，产生`ty` 类型的结果。它是通过使用`runState`、`execState`或`evalState`之一执行的。

许多有趣的程序都遵循这种模式，定义了一个类型来描述

命令序列和用于执行这些命令的单独函数。

事实上，你已经在第 11 章中看到了一个，当你定义了`ConsoleIO`类型时

用于描述无限期运行的交互式程序。你会看到更多的例子

剩下的章节，所以在本章的其余部分，我们将看看如何实现

用于表示状态和交互的自定义类型。

练习



1 编写一个函数，通过将函数应用于当前状态来更新状态：

更新：(状态类型->状态类型) ->状态状态类型 ()

您应该能够使用`update`重新实现增加：

增加：Nat -> State Nat ()

增加 $x = \text{更新 } (+x)$

您可以在REPL上测试您的答案，如下所示：

*ex_12_1> 运行状态 (增加 5)89

((), 94) : ((), 纳特)

2 编写一个函数，使用`State`计算`Empty`在

树。它应该具有以下类型：

countEmpty : 树 a -> 状态 Nat ()

您可以使用`testTree`在REPL测试您的答案，如下所示：

*ex_12_1> execState (countEmpty testTree) 0
7.纳特

3 编写一个函数，计算一个函数中`Empty`和`Node`出现的次数

树，使用`State`存储一对中每个的计数。它应该有以下内容
类型：

countEmptyNode : 树 a -> 状态 (Nat, Nat) ()

您可以使用`testTree`在REPL测试您的答案，如下所示：

*ex_12_1> execState (countEmptyNode testTree) (0, 0)
(7, 6) : (晚上, 晚上)

12.2 State 的自定义实现

在上一节中,您看到State类型

为您提供了实现使用状态的算法的通用方法。您使用它将标签流维护为本地可变状态,您可以根据需要通过读取 (使用get)和写入 (使用put)来访问它。就像IO将交互式程序的描述与其在运行时的执行分开一样, State将有状态程序的描述与其具有具体状态的执行分开。

在剩下的章节中,我们将看到更多相同模式的示例,将程序的描述与其执行分开,所以在我继续之前,让我们探索如何自己定义State类型,以及用于exe的runState削减有状态的操作。在本节中,您将看到定义State的一种方式,以及如何为State 提供一些接口的实现: Functor、 Applicative和Monad。通过实现这些接口,您将能够在State 中使用一些通用库函数。

12.2.1 定义状态和运行状态

下面的清单显示了一种手动定义State的方法,使用Get数据构造函数描述读取状态的操作,使用Put数据构造函数描述写入状态的操作。

清单 12.8 描述有状态操作的类型 (TreeLabelText.idr)

```
数据状态: (stateType:Type) -> Type-> Type where
    获取:状态 stateType stateType
    放: stateType -> 状态 stateType ()
    纯:ty -> 状态 stateType ty
    绑定:状态 stateType a -> (a -> State stateType b) -> State stateType b

    产生值的操作
    描述获取当前状态的操作
    描述设置新状态的操作
    对有状态操作进行排序,将第一个的结果作为输入传递给下一个
```

命名约定提醒请记住,按照约定,
Idris 中的类型和数据构造函数名称以大写字母开头。我不会在这里偏离这个约定,所以如果想要与 Control.Monad.State 导出的名称相同的名称,您需要定义以下函数:

```
获取:状态 stateType stateType get = 获取
put : stateType -> 状态 stateType () put = Put
pure : ty -> 状态 stateType ty pure = Pure
```

您可以通过定义($>=$)来支持State的do表示法。您可以通过实现($>=$)的Monad接口或直接定义($>=$)来做到这一点：

```
(>=) : State stateType a -> (a -> State stateType b) -> State stateType b
(>=) = 绑定
```

为状态定义 ($>=$) 实现接口意味着您可以对状

态中的程序使用 do 表示法。正如你在第 7 章中看到的,($>=$) 也是 Monad 接口的一个方法,它也需要 Functor 和 Applicative 接口的实现。它在这里被定义为一个独立的函数,以避免在这个例子中首先需要实现 Functor 和 Applicative。

在可能的情况下,实现 Monad 接口是一个非常好的主意,因为这使您可以访问 Idris 库定义的几个受约束的泛型函数。例如,您可以使用 when 函数 (在满足条件时执行操作) 和 traverse (跨结构应用计算)。您将在第 12.2.2 节中看到如何为 State 执行此操作。

使用这个版本的State,并定义函数get、put和pure,它们直接使用数据构造函数,清单 12.9 展示了如何定义treeLabel With。如您所料,此版本与前一个版本完全相同,因为它对操纵状态的函数使用相同的名称。

清单 12.9 将treeLabelWith 定义为一系列有状态操作 (TreeLabelText.idr)

```
treeLabelWith : Tree a -> State (Stream labelText) (Tree (labelType, a))
treeLabelWith Empty = Pure Empty
treeLabelWith (Node left val right) = do left_labelled <- treeLabelWith left (this :: rest) <- get
    put rest right_labelled <- treeLabelWith right pure (Node left_labelled (this, val) right_labelled)
    return (left_labelled, right_labelled)
```

为了运行它,您需要定义一个函数,将有状态操作的描述转换为树标签函数。下面的清单显示了runState 的定义,它接受有状态程序的描述和初始状态,并返回有状态程序产生的值和最终状态。

清单 12.10 运行标签操作 (TreeLabelText.idr)

```
runState : State stateType a -> (st : stateType) -> (a, stateType)
runState Get st = (st, st)
runState (Put newState) st = ((), newState)
```

状态的自定义实现

```

runState (Pure x) st = (x, st)
runState (Bind cmd
prog) st = let (val, nextState) = runState cmd st in
            → runState (prog void) nextState
运行第一个有状态命令,然后使用更新的状态和第一个命令
的输出运行程序的其余部分

```

当您运行使用Bind定义的有状态操作序列时,您需要获取运行cmd返回的nextState状态,并在执行其余操作时将其传递给runState。您可以通过获取运行cmd返回的val并将其传递给prog来计算其余操作。这封装了您在第一次实现treeLabelWith时必须手动实现(三次!)的状态管理,它类似于Control.Monad.State模块实现State本身的方式。

下面的清单显示了treeLabel的定义,它使用treeLabelWith的新实现,用[1..]初始化流。

**清单 12.11 树标签函数,它调用run时带有一个初始的标签流
(TreeLabelText.idr)**

```
treeLabel : 树 a -> Tree (Integer, a) treeLabel tree = fst (runState
(treeLabelWith tree) [1..])
```

← 使用 fst 提取标记树并丢弃
最终状态

12.2.2 为 State 定义 Functor、Applicative 和 Monad 实现

为State实现($>>=$)意味着您可以使用do表示法,它为编写描述操作序列的函数提供了清晰易读的表示法。

但是做符号是它给我们的全部。

与其将($>>=$)定义为独立函数,不如为State实现Functor、Applicative和Monad接口。除了通过Monad接口提供do表示法之外,这将使您能够访问库中定义的通用函数。例如,when和traverse是通用函数。在IO的上下文中,它们的行为如下:

当条件为真时评估计算。它可以用来运行
某些IO操作仅针对特定用户输入。
traverse类似于map并跨结构应用计算。例如,您可以将List的每个元素打印到控制台。

您可以使用:doc了解有关这些函数的更多信息,尤其是它们的类型。下面的清单显示了它们在IO计算的上下文中的作用。

清单 12.12 使用when和traverse (Traverse.idr)

```
船员:列表字符串船员= [ “Lister” ,
“Rimmer” , “Kryten” , “Cat” ]
```

```
main : IO () main =
do putStrLn "Display Crew? "
x <- getLine when (x ==
    yes ) $ do traverse putStrLn crew
pure() putStrLn "Done"
```

仅当此条件为真时才评估 \$ 之后的计算。\$ 是应用程序运算符。

对于工作人员列表中的所有内容,评估 putStrLn

应用运算符 \$ 请记住,在第 10 章中,\$ 运算符是一个中缀运算符,它将函数应用于参数。其主要目的是减少对包围的需要。在清单 12.12 中,您还可以使用显式括号编写 when 的应用程序,如下所示:

```
当 (x == yes ) (做遍历
    putStrLn 船员纯 ())
```

如果您为State实现Functor、Applicative和Monad,您将能够在使用State的函数中使用这些和其他类似的函数。下一个清单显示了您可以做什么的示例,提供了一个将列表中的整数添加到运行总数的函数,前提是整数是正数。目前,这将无法进行类型检查。

清单 12.13 将列表中的正整数添加到状态 (StateMonad.idr)

返回整数是否添加成功

```
addIfPositive : Integer -> State Integer Bool addIfPositive val = do when (val
> 0) $
    makeCurrent <- get
    modify (makeCurrent + val)
```

如果为正,则使用给定值递增状态

纯 (val > 0)

```
addPositives : List Integer -> State Integer Nat addPositives vals = do added <-
    traverse addIfPositive vals
```

纯 (length (添加过滤器ID))

对于 vals 中的每个整数, added 中的值对应于该整数是否已添加到状态中。

添加的过滤器 id 给出了添加的元素为真,因此这将返回成功添加的整数的数量。

这将失败,因为您没有Functor或Applicative的实现

状态:

StateMonad.idr:42:15:当检查
addIfPositive 的右侧与预期类型时
状态整数布尔

检查函数 Main.>>= 的应用程序时,找不到 Applicative (State Integer) 的实现

您在这里的最终目标是实现Monad for State,这也需要实现Functor和Applicative。下一个清单显示了在 Prelude 中定义的Monad接口。

清单 12.14 Monad接口

```
interface Applicative m => Monad (m : Type -> Type) where (>>=) : ma -> (a -> mb) -> mb join :
    m (ma) -> ma
    ↗          ↘
    “扁平化”嵌套结构。请参阅侧边栏。
    将第一个操作的输出作为输入传递给第二个操作
```

join 方法

我们没有详细研究 join,但是您可以使用它将嵌套结构扁平化为单个结构。例如,有 List 和 Maybe 的 Monad 实现,因此您可以尝试加入每个示例:

伊德里斯> 加入 [[1,2,3], [4,5,6]]
[1, 2, 3, 4, 5, 6] :列表整数

伊德里斯>加入 (只是 (只是 “一”))
只是 “一个” :也许是字符串

伊德里斯>加入 (只是 (无{a =字符串}))
什么都没有:也许是字符串

对于 List.join 将连接嵌套列表。对于 Maybe.join 将查找嵌套在结构中的单个值 (如果有)。

(>>=)和join两种方法都有默认定义,因此您可以通过定义其中一个或两个来实现Monad。在这里,我们将只使用(>>=)。

如果要为Monad 提供实现,还需要实现Applicative,因为它是Monad 的父接口。同样, Applicative 有一个父接口Functor。以下清单显示了 Prelude 中定义的两个接口。

清单 12.15 Functor和Applicative接口

```
interface Functor (f : Type -> Type) where map : (func : a -> b) -> fa ->
    fb
    ↗
    接口 Functor f => Applicative (f : Type -> Type) 其中
    纯:a -> fa (<*>): f (a -> b) ->
    fa -> fb
    ↗
    将函数应用于参数,其中函数和参数位于结构内
```

(*<*>*)方法允许您拥有一个返回函数 (*a -> b* 类型)的有状态函数,拥有另一个返回参数 (*a* 类型)的有状态函数,并将该函数应用于论点。

您可以从实现Functor开始,如下所示: 1键入,定义 编写实现标头并创建映射的骨架定义。请记住,您可以通过将光标放在接口名称上按 Ctrl-Alt-A 来为接口方法创建骨架定义:

```
函子 (State stateType) where map func x = ?Functor_rhs_1
```

2类型,细化 - ?Functor_rhs_1孔的类型告诉您x是有状态的计算:

```
stateType : 类型 b : 类型 a : 类
型 func : a -> b
```

```
x : 状态 stateType a
-----
Function_rhs_1 : 状态 stateType b
```

您可以通过使用Bind从计算x中提取值来继续定义:

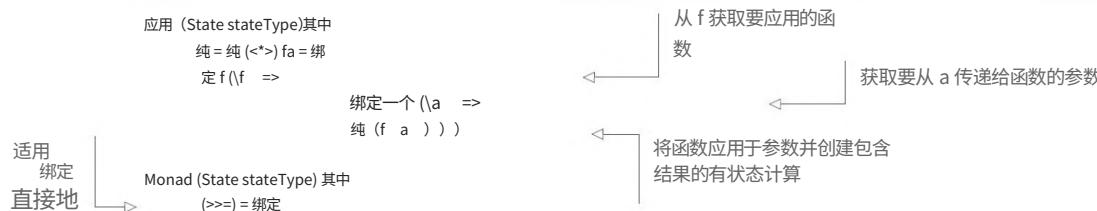
```
Functor (State stateType) where map func x = Bind x
(\val => ?Functor_rhs_1)
```

3 Refine 要完成定义,将val传递给func,并使用Pure将结果转换为有状态计算:

```
Functor (State stateType) where map func x = Bind x
(\val => Pure (func val))
```

清单 12.16 显示了Applicative和Monad的State 定义。您以类似于Functor的方式实现Applicative , 使用Bind从有状态计算中提取您需要的值。

清单 12.16为State实现Applicative和Monad (StateMonad.idr)

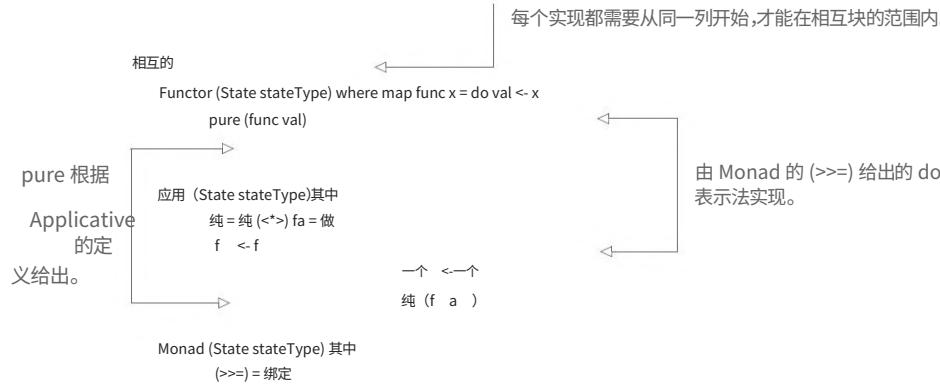


您在Functor和Applicative的实现中使用了Bind ,因为您还没有可用的do表示法。你需要一个Monad实现来提供它,

你需要有Functor和Applicative实现才能有Monad实现。

但是您可以通过在一个相互块中一起定义所有实现来使用do表示法,如清单 12.17 所示。在相互块中,定义可以相互引用,因此Functor和Applicative的实现可以依赖于Monad 的实现。

清单 12.17 一起定义Functor、 Applicative和Monad实现 (StateMonad.idr)



现在您已经实现了这些接口,您可以尝试清单 12.13中addPositives的早期定义:

```
*StateMonad> runState (addPositives [-4, 3, -8, 9, 8]) 0
(3, 20) : (Nat, 整数)
```

Effects 库:结合 State、IO 和其他副作用 鉴于您有 State 和 IO 的 Monad 实现,分别对有状态计算和交互式计算进行排序,有理由想知道您是否可以在同一个函数中同时对两者进行排序 什么关于也操纵状态的交互式程序?

您将在下一节中看到一种方法。不过,作为一个更通用的解决方案,Idris 提供了一个名为 Effects 的库,它支持组合不同类型的副作用,例如类型中的 State 和 IO,以及其他效果,例如异常和不确定性。您可以在效果库教程 (<http://idris-lang.org/documentation/effects>) 中找到更多详细信息。

您现在已经了解了如何通过将有状态操作序列描述为状态并使用 runState 执行它们来封装状态操作的细节。您还看到了如何为State实现Functor、 Applicative和Monad。

到目前为止,您在示例中使用的状态相当小:单个流
标签或单个整数。更一般地说,状态会变得相当复杂:

状态可能由几个复杂的组件组成,存储为记录。我们将在本章的其余部分讨论这个问题,
在那里你将看到如何编写一个完整的
带有状态的交互式程序。

您可能希望在您的状态中使用依赖类型,在这种情况下更新
状态也会更新它的类型!例如,如果您将元素添加到Vect
8 Int,它将成为Vect 9 Int。我们将在下一章讨论这个问题。

编写表达命令序列的类型以及
运行这些命令的功能,是您可以将命令类型设置为
根据需要精确。正如您将在下一节中看到的,您可以精确地描述集合
特定应用程序所需的命令,包括交互命令
在控制台和用于读取和写入应用程序状态组件的命令。在下一章中,您将看到如何在其类型中精确描述
每个命令对系统状态的影响。

12.3 带状态的完整程序:使用记录

在上一章中,您实现了一个算术测验,向用户展示了乘法问题,并记录了正确答案的数量和

问的问题。在本节中,我们将编写一个精炼的版本,具有以下内容
改进:

我们将添加一个难度设置,它指定了当
产生问题。
不是将当前分数作为参数传递给quiz函数,而是手动维护状态,我们将使用命令扩展Command类
型,以访问当前分数和难度设置。

要编写这个精炼的版本,我们需要重新考虑如何表示游戏的状态。
我们将使用记录类型来做到这一点。你已经在第 6 章看到了一些使用记录来表示数据存
储的方法,在第 7 章和第 10 章中也有类似的例子。
但是,应用程序的状态会增长,将其状态划分为几个层次结构记录是有意义的。

您将看到如何定义和使用嵌套记录,如何使用简洁的语法更新记录,以及如何使用记录来存储交互式
测验程序的状态。
不过,首先,我们将重温第 11 章中的Command类型,看看如何
以类似于自定义的方式扩展它以支持读写系统状态
上一节中定义的状态类型。

12.3.1 带状态的交互式程序:重新审视算术测验

在第 11 章中,您定义了一个Command数据类型,表示您可以执行的命令
在控制台 I/O 程序中使用,以及一个ConsoleIO类型,表示可能是无限的
交互过程。你用它来实现一个算术测验,展示

带状态的完整程序:使用记录

乘法问题供用户回答。与描述读取和写入状态的Get和Put操作的State一样，Command描述用于读取和写入控制台的Get Line和PutStr操作。以下清单

回顾Command和ConsoleIO的定义。

清单 12.18 仅支持控制台 I/O 的交互式程序 (ArithState.idr)

```

数据命令 : 类型 -> 键入位置
PutStr : 字符串 -> 命令 ()
GetLine : 命令字符串

纯 : ty -> 命令 ty
绑定 : 命令 a -> (a -> 命令 b) -> 命令 b

数据 ConsoleIO : 类型 -> 在哪里键入
退出 : a -> ConsoleIO a
执行 : 命令 a -> (a -> Inf (ConsoleIO b)) -> ConsoleIO b

命名空间 CommandDo
(>>=) : 命令 a -> (a -> 命令 b) -> 命令 b
(>>=) = 绑定

命名空间 ConsoleDo
(>>=) : 命令 a -> (a -> Inf (ConsoleIO b)) -> ConsoleIO b
(>>=) = 做

```

为命令实现MONAD与State一样,您可以实现
Functor、Applicative和Monad用于命令而不是实现
(>>=)直接。作为练习,尝试提供每个的实现。正如我
然而,在上一章中提到,您不能为ConsoleIO提供Monad实现,因为ConsoleDo.(>>=)的类型
不适合。

像runState,它接受有状态操作的描述并返回结果

在以初始状态执行这些操作时, run获取交互操作的描述并在IO中执行它们。清单 12.19 概括了run函数。

请记住,您通过使用Fuel限制了交互式程序可以运行的时间
类型,然后添加一个非全函数,永远,它允许一个全交互程序
无限期地运行,同时只引入一个非全函数。

清单 12.19 运行交互式程序 (ArithState.idr)

```

数据燃料 = 干燥 | 更多 (懒惰的燃料)

永远 : 燃料
永远 = 永远更多

runCommand : 命令 a -> IO a
runCommand (PutStr x) = putStr x
运行命令 GetLine = getLine
runCommand (纯验证) = 纯验证
runCommand (Bind cf) = do res <- runCommand c
    运行命令 (f res)

```

```

run : Fuel -> ConsoleIO a -> IO (Maybe a) run fuel (Quit val) = do pure (Just val)
run (More fuel) (Do cf) = do res <- runCommand c
                                         ↑
                                         只要有燃料,就执行交互
                                         式程序
                                         ↓
                                         运行燃料 (f res)

运行 Dry p = pure Nothing

```

如果您希望您的交互式程序能够读取和写入状态,除了从控制台读取和写入之外,您还可以使用其他命令扩展Command类型来操作状态,并像处理State一样处理这些命令.对于算术游戏,我们需要执行以下操作:

给定游戏的难度设置,获取问题的随机数。读取当前游戏状态,以便显示分数。更新游戏状态,以便根据用户的情况更新分数

答案。

下一个清单显示了如何扩展Command数据类型以包含这些命令。无需更新ConsoleIO,因为它只是对命令进行排序。

清单 12.20 扩展Command类型以支持游戏状态 (ArithState.idr)

```

GameState:类型
                                         ←
                                         这是一个占位符。您将很快
                                         定义 GameState。
数据命令:类型 -> 键入位置
PutStr : 字符串 -> 命令 ()
GetLine : 命令字符串
GetRandom:命令整数
GetGameState : 命令 GameState
PutGameState:GameState -> 命令 ()
                                         ←
                                         返回一个随机数,基于
                                         游戏当前的难度级别
                                         ←
                                         返回当前游戏状态
                                         ←
                                         设置游戏状态
纯: ty -> 命令 ty
绑定:命令 a -> (a -> 命令 b) -> 命令 b

```

GameState暂时未定义,因此您还不能完成run或runCommand。但是,您可以为额外的命令添加带有孔的模式子句,以便满足整体检查器的要求:

```

runCommand : 命令 a -> IO a
runCommand (PutStr x) = putStr x runCommand GetLine =
getLine runCommand (Pure val) = pure val runCommand
(Bind cf) = do res <- runCommand c
                                         ↓
                                         运行命令 (f res)
runCommand (PutGameState x) = ?runCommand_rhs_1 runCommand
GetGameState = ?runCommand_rhs_2 runCommand GetRandom = ?
runCommand_rhs_3

```

添加遗漏的情况将构造函数添加到Command后,您可以使用 Ctrl-Alt-A 并将光标悬停在类型声明中的runCommand上,为 runCommand 添加新的模式子句。

带状态的完整程序 : 使用记录

您将使用GameState类型来存储游戏的状态。因此,在实施精炼测验之前,您需要考虑如何定义GameState。

12.3.2 复杂状态 : 定义嵌套记录

在优化的测验实现中,您将使用GameState来存储以下内容:

- 当前分数,由正确回答的问题数组成
- 和尝试次数难度设置

当这样的程序状态有多个组件时,使用记录类型通常是有意义的。记录很方便,因为它们会产生投影功能,使您可以检查记录的字段。您还可以嵌套记录;以下清单显示了如何将当前分数表示为记录,将整体游戏状态表示为记录,包括作为字段的嵌套分数记录。

清单 12.21 将游戏状态表示为嵌套记录 (ArithState.idr)

```

记录分数在哪里
    构造函数 MkScore
    正确: 布尔
    尝试:Nat

记录 GameState 在哪里
    构造函数 MkGameState
    分数:分数
    难度:智力
initState : GameState initState =
MkGameState (MkScore 0 0) 12

```

当前分数包含正确答案和尝试问题数量的字段

游戏状态记录包括得分记录作为该记录中的一个字段。

初始游戏状态为 0 分 (满分 0), 难度为 12。

将Score和GameState定义为记录会自动为每个字段生成投影函数:正确、尝试、得分和难度。例如,您可以像这样获得难度级别:

```
*ArithState> 难度initState
12 : 诠释
```

或者您可以获得到目前为止正确答案的数量:

```
*ArithState> 正确 (得分 initState)
0 : 自然
```

记录和名称空间 定义记录时,投

影函数在它们自己的名称空间中定义,由记录的名称给出。例如,在新的 GameState 命名空间中定义了 score 函数,正如您可以在 REPL 中使用 :doc 看到的那样:

```
*ArithState> :doc 分数
Main.GameState.score : (rec : GameState) -> 分数
```

(继续)

这允许在同一模块中多次使用相同的字段名称。
例如,您可以在同一文件 Record.idr 的两个不同记录中使用名为 title 的字段:

```
记录书在哪里
构造函数 MkBook
标题:字符串作者:字符串
```

```
记录专辑在哪里
构造函数 MkAlbum
标题:字符串曲目:列出字符串
```

Idris 将根据上下文决定使用哪个版本的标题:

```
*记录>标题 (MkBook "冠军早餐" "库尔特·冯内古特")
《冠军的早餐》:弦乐
```

以下清单显示了如何使用投影函数来定义GameState的Show实现。

清单 12.22 显示 GameState (ArithState.idr) 的实现

```
在哪里显示 GameState
显示 st = 显示 (正确 (分数 st)) ++ "/" ++
显示 (尝试 (得分 st)) ++ "\n" ++
"难度：" ++ 显示 (难度 st)
```

您可以在REPL 上尝试此操作,使用println显示初始游戏状态:

```
*ArithState> :exec println initState 0/0
```

```
难度:12
```

因此,记录为您检查字段值提供了一种方便的表示法,但是当您编写使用记录来保存状态的程序时,您还需要更新字段。例如,在测验中,您需要增加分数和尝试的问题数量。

12.3.3 更新记录字段值

Idris 是一种纯函数式语言,因此您不会就地更新记录字段。相反,当我们说我们正在更新一条记录时,我们的真正意思是正在构建一条新记录,其中包含旧记录的内容,其中一个字段已更改。例如,以下清单显示了一种使用模式匹配返回更新难度字段的新记录的方法。

带状态的完整程序：使用记录

清单 12.23 通过模式匹配设置记录字段 (ArithState.idr)

```
setDifficulty : Nat -> GameState -> GameState
setDifficulty newDiff (MkGameState
score _) = MkGameState score newDiff
```

如果记录有很多字段，这会很快变得笨拙，因为您需要为每个字段编写更新函数。不仅如此，如果要向记录添加字段，则需要修改所有更新函数。因此，Idris 提供了一个内置的语法来更新记录中的字段。这是使用记录更新语法的 setDifficulty 的实现。

清单 12.24 使用记录更新语法设置记录字段 (ArithState.idr)

```
setDifficulty : Nat -> GameState -> GameState
setDifficulty newDiff 状态 = 记录 { 难度 =
newDiff } 状态
```

图 12.3 显示了记录更新语法的组成部分。特别注意，记录更新语法本身是一等的，其中记录是开始记录更新的关键字，因此记录更新具有类型。这里，更新有一个函数类型，`GameState -> GameState`，所以也可以实现 setDifficulty，如下：

```
setDifficulty : Nat -> GameState -> GameState
setDifficulty newDiff = 记录 { 难度 =
newDiff }
```



图 12.3 返回带有更新字段的新记录的语法

您可以以类似的方式更新嵌套记录字段，方法是提供要更新的字段的路径。下面的清单显示了如何编写正确和错误的函数来更新分数。

清单 12.25 使用记录更新语法设置嵌套记录字段 (ArithState.idr)

```
addWrong : GameState -> GameState
addWrong state = record
{ score->attempted = mapped (score state)+1 } state
```

```
addCorrect : GameState -> GameState
添加正确状态
```

通过在当前值上加 1 来更新尝试的问题

数

```
= 记录 { 分数->正确 = 正确 (分数状态)+ 1,
          score->attempted = 已尝试 (得分状态)+1} 状态
```

通过在当前值上加 1 来更新正确答案的数量

您可以一次更新多个字段，用逗号分隔更新。

`score->attempted` 表示法给出了您要更新的字段的路径，其中最外层的字段名在前。所以，在这个例子中，你想更新尝试的状态记录的 `score` 字段的字段。

12.3.4 应用函数更新记录字段

记录更新语法为指定特定记录字段的路径提供了一种简洁的符号。不过还是有点不方便，因为你需要

以不同的符号显式将每个字段的路径写入两次：

首先，找到要更新的字段，使用 `score->correct` path notation
二、找旧值，使用函数应用，`正确 (分数状态)`

因此，Idris 通过直接应用
函数为字段的当前值。下一个清单显示了一种简洁的方式
更新 `GameState` 中的嵌套记录字段。

清单 12.26 通过直接应用函数来更新嵌套记录字段 字段的当前值 (`ArithState.idr`)

```
addWrong : GameState -> GameState
addWrong = 记录 { score->attempted $= (+1) }

addCorrect : GameState -> GameState
addCorrect = 记录 { 分数->正确 $= (+1),
                      得分->尝试 $= (+1) }
```

字段更新中的 `$=` 运算符表示通过将函数 `(+1)` 应用于当前值来计算新的字段值。

使用 `$=` 更新记录您看到了 `$` 运算符，它应用了一个函数到一个论点，在第 10 章。`$=` 语法来自于函数应用运算符 `$` 和记录更新语法。

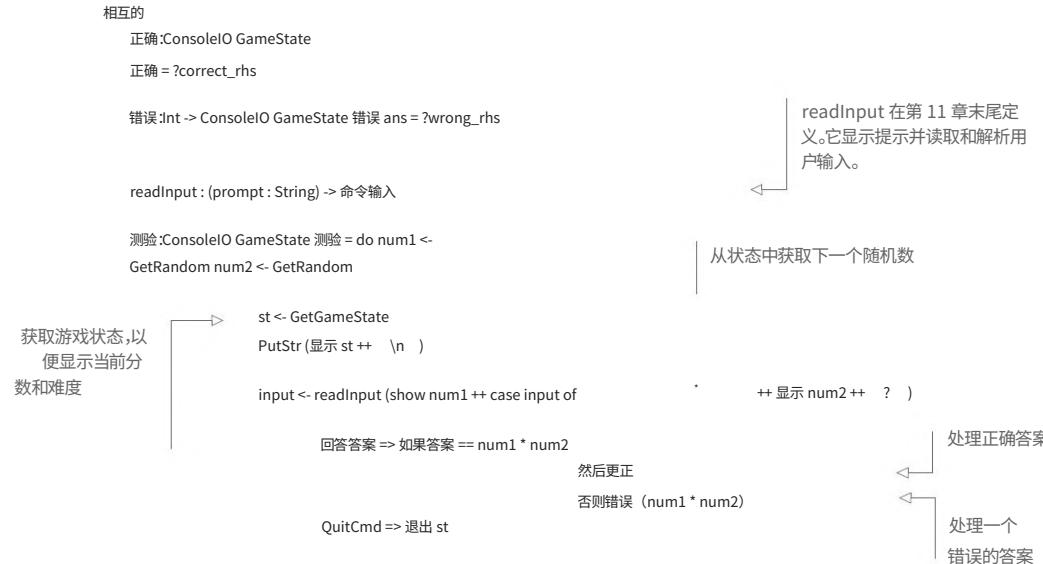
此语法为您提供了一种简洁方便的方法来编写更新函数
嵌套记录字段，这使得编写操作状态的程序更容易。
此外，因为它不使用模式匹配，它独立于任何其他
记录中的字段，因此即使您将字段添加到 `GameState` 记录，`addWrong` 和
`addCorrect` 无需修改即可工作。

12.3.5 实施测验

使用新的 `Command` 类型和 `GameState` 记录，您可以根据需要通过读取和更新状态来实施算术测验。下一个清单显示了一个测验实施的大纲，留下正确和错误的洞，其中，分别处理一个正确答案和一个错误答案。

带状态的完整程序:使用记录

清单 12.27 实现算术测验 (ArithState.idr)



这类似于第 11 章末尾的quiz实现,但不是将随机数流和分数作为参数传递,而是将它们中的每一个视为您根据需要读取和写入的状态。这简化了quiz 的定义,但代价是使ConsoleIO的定义更加复杂。

下一个清单显示了如何实现正确和错误,每个修改状态分别使用addCorrect和addWrong,如上一节中所定义。

清单 12.28 通过更新游戏状态 (ArithState.idr) 处理正确和错误的答案



在这个阶段,您对交互式算术测验有完整的描述,该测验从状态中检索随机数和当前分数。也是总数:

*ArithState> :total quiz Main.quiz 是 Total

但是要运行测验,您需要扩展runCommand以支持您的新命令。

12.3.6 运行交互式和有状态程序:执行测验

与您为自定义中的处理操作编写的runState函数一样

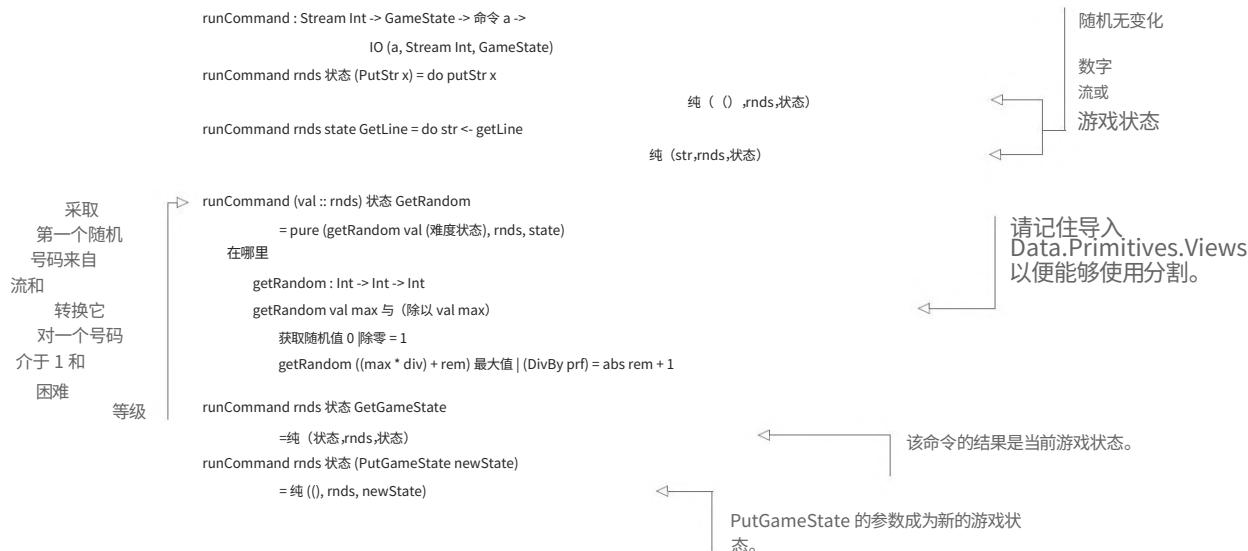
12.2 节中的state类型,更新后的run函数将需要处理当前的游戏状态。它还需要从随机数流中读取 (用于获取随机数)并执行控制台I/O。以下清单显示了如何将这些全部捕获到runCommand 的新类型中。

清单 12.29 runCommand (ArithState.idr)的新类型和骨架定义



清单 12.30 给出了runCommand 的完整定义。请注意,在每种情况下,您需要返回命令的结果以及显示每个命令如何影响随机数流和游戏状态。

清单 12.30 runCommand (ArithState.idr)的完整定义



带状态的完整程序 : 使用记录



同样,您需要更新run以获取随机整数流和初始游戏状态。与runCommand一样, run也返回运行程序的结果,以及更新的流和游戏状态。这是运行的新实现,它支持游戏状态。

清单 12.31 运行一个包含可能无限的Command流(ArithState.idr)的ConsoleIO程序

正如在第 11 章中,你使用Fuel来表示你愿意允许一个潜在的无限交互程序运行多长时间,所以返回类型中表示程序运行结果的部分,类型为ConsoleIO a,类型为Maybe a,捕捉燃料耗尽的可能性。

最后,下一个清单显示了如何更新主程序以初始化随机数流和游戏状态。

**清单 12.32 一个用随机数初始化算术测验的主程序
流和初始状态 (ArithState.idr)**

←—— 生成无限的随机数流

←—— 需要,因为您永远使用

←—— 记得导入系统才能使用时间。

与第 11 章末尾的quiz实现一样,您将命令的终止序列(使用Command类型)与可能的非终止的控制台I/O 操作序列(使用ConsoleIO类型)分开。此外,您扩展Command类型以允许读取和写入游戏状态,就像

国家的实施。如果您为命令定义特定类型
 应用程序可以执行,您可以根据需要使该类型尽可能精确
 描述每个操作对系统状态抽象的影响。你会看到的
 通过遵循这种模式和保证可以实现的更多目标
 在下一章中讨论有状态的交互式程序。

练习

1 编写具有以下类型的updateGameState函数:

```
updateGameState : (GameState -> GameState) -> Command()
```

您可以在正确和错误的定义中使用它来测试它,而不是Get GameState和PutGameState。
 例如:

```
正确:ConsoleIO GameState
正确 = 执行 PutStr "正确!\n"
      updateGameState 添加正确
      测试
```

2 为Command实现Functor、Applicative和Monad接口。

3 您可以将用于在社交新闻网站上表示一篇文章的记录定义如下,以及文章被赞成或反对的次数:

```
在哪里记录投票
构造函数 MkVotes
赞成票:整数
否决票:整数

记录文章在哪里
构造函数 MkArticle
标题:字符串
网址:字符串
得分:投票

initPage : (title : String) -> (url : String) -> 文章
initPage 标题 url = MkArticle 标题 url (MkVotes 0 0)
```

编写一个函数来计算给定文章的总分,其中得分为
 由赞成票数减去反对票数计算得出。
 它应该具有以下类型:

```
getScore : 文章 -> 整数
```

您可以使用以下示例文章对其进行测试:

```
badSite : 文章
badSite = MkArticle "坏页" "http://example.com/bad" (MkVotes 5 47)

goodSite : 文章
goodSite = MkArticle "好页面" "http://example.com/good" (MkVotes 101 7)
```

在REPL中,您应该看到以下内容:

```
*ex_12_3> getScore goodSite
94:整数
```

```
*ex_12_3> getScore badSite
-42:整数
```

4 编写addUpvote和addDownvote函数来修改文章的得分或下。它们应具有以下类型：

```
addUpvote : 文章 -> 文章 addDownvote : 文章 -> 文章
```

您可以在REPL上进行如下测试：

```
*ex_12_3> addUpvote goodSite MkArticle “好
页面” “http://example.com/good”
```

```
(MkVotes 102 7) :文章
```

```
*ex_12_3> addDownvote badSite MkArticle
Bad Page http://example.com/bad
```

```
(MkVotes 5 48) :文章
```

```
*ex_12_3> getScore (addUpvote goodSite)
95:整数
```

12.4 总结

许多算法读写状态。例如，当以深度优先顺序标记树的节点时，您可以在遍历树时跟踪标签的状态。

您可以通过使用通用State类型来描述对状态的操作以及执行这些操作的runState函数来管理状态。您可以将自己
的State类型定义为一系列Get和Put操作。为State类型定义Functor、Applicative和Monad使您可以访问
Idris 库中的几个通用函数。在编写带状态的交互程序时，可以定义一个Command类型来描述一个由控制台I/O和
状态管理操作组成的接口。记录类型可以表示更复杂的嵌套状态。Idris 提供了一种简洁的语法来为嵌套中的字
段分配新值

记录。

Idris 还提供了一种语法（使用\$=），用于通过以下方式更新嵌套记录中的字段
将函数应用于该字段中的值。

状态机：验证类 型中的协议

本章涵盖

在类型中指定协议描述操作的前置
条件和后置条件在状态中使用依赖类型

在上一章中，您了解了如何通过定义表示系统中命令序列的类型以及运行这些命令的函数来管理可变状态。这遵循一个常见模式：数据类型描述一系列操作，函数在特定上下文中解释该序列。例如，`State`描述有状态操作的序列，而`runState`解释具有特定初始状态的操作。

在本章中，我们将看到使用类型来描述操作序列和保持执行功能分开的优点之一。它允许您使描述更精确，以便某些操作只能在状态具有特定形式时运行。例如，某些操作在执行之前需要访问资源，例如文件句柄或数据库连接：

您需要一个打开的文件句柄才能成功读取文件。
在对数据运行查询之前,您需要连接到数据库
根据。

当您编写使用此类资源的程序时,您实际上是在使用
一个状态机。一个数据库客户端可能有两种状态,比如关闭和连接,
指其与数据库的连接状态。一些操作 (例如查询
database) 仅在Connected状态下有效;有些 (如连接数据库) 只在Closed状态下有效;还有一些 (比如
连接和关闭)
也改变了系统的状态。图 13.1 说明了这个系统。

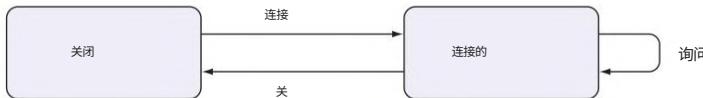


图 13.1 显示数据库高级操作的状态转换图。它有两种可能的状态,关闭和连接。它的三个操
作,连接、查询和关闭,仅在特定状态下有效。

图 13.1 所示的状态机隐含地存在于许多现实世界的系统中。例如,当您实施通信系统时,

无论是通过网络还是使用并发进程,您都需要确保每个
方遵循相同的通信模式,否则系统可能会死锁或
以其他意想不到的方式行事。每一方都遵循一个状态机,在该状态机中发送或接收消息会将整个系统置于
一个新状态,因此这很重要
每一方都遵循明确定义的协议。在 Idris 中,我们有一个富有表现力的类型
系统,所以如果有一个协议模型,最好用一个类型来表达它,所以
您可以使用该类型来帮助准确地实现协议。

在本章中,您将看到如何使状态机像图13.1所示的状态机在类型中显式。这样,您可以确定任何正确的
功能
描述遵循状态机定义的协议的一系列动作。不是
仅此而已,您可以采用类型驱动的方法来定义操作序列,使用
漏洞和互动发展。我们将从一些相当抽象的例子开始
说明如何以类型描述状态机,对门和自动售货机上的状态和操作进行建模。

13.1 状态机 :在类型中跟踪状态

您之前通过定义描述的类型来实现具有状态的程序
用于读取和写入状态的命令。使用依赖类型,您可以使
这些命令的类型更精确,并包括有关状态的任何相关详细信息
类型本身的系统。

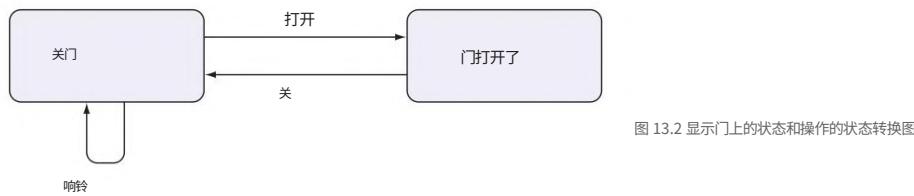
例如,让我们考虑如何用门铃来表示门的状态。一个
门可以处于两种状态之一,打开 (表示为DoorOpen) 或关闭 (表示为 DoorClosed), 我们将允许三个
操作:

打开门,将系统从DoorClosed状态移动到
开门状态

关门,将系统从DoorOpen状态移动到Door
关闭状态

敲门铃,只有当门在门中时才允许
关闭状态

图 13.2 是一个状态转换图,显示了系统可以处于的状态以及每个操作如何修改整体状态。



如果您可以在一个类型中定义这些状态转换,那么对操作序列的良好类型化描述必须正确遵循状态转换图中显示的规则。此外,您将能够使用孔洞和交互式编辑来找出在序列中的特定点哪些操作是有效的。

在本节中,您将看到如何在依赖类型中定义状态机,例如门。首先,我们将实现门的模型,然后我们将在简化的自动售货机模型中对更复杂的状态进行建模。在每种情况下,我们都将关注状态转换的模型,而不是机器的具体实现。

13.1.1 有限状态机:将门建模为类型

图 13.2 中的状态机通过说明哪些操作在哪个状态下有效,以及这些操作如何影响状态来描述正确使用门的协议。

清单 13.1 展示了一种表示可能操作的方法。这还包括用于排序的($>=$)构造函数和用于生成纯值的Pure构造函数。

清单 13.1 将门上的操作表示为命令类型 (Door.idr)

```

数据 DoorCmd :键入 where
  打开 : DoorCmd ()
  关闭:DoorCmd ()
  响铃:DoorCmd ()

纯:t -> DoorCmd ty
(>=) : DoorCmd a -> (a -> DoorCmd b) -> DoorCmd b
  
```

↓ 将门的状态从 DoorClosed 更改
为 DoorOpen

↓ 将门的状态从 DoorOpen 更改为
DoorClosed

提醒: ($>=$) 和 DO 表示法请记住, do 表示法转化为($>=$) 的应用。

使用DoorCmd,您可以编写如下函数,它描述了按门铃和打开然后关闭门的一系列操作,正确地遵循门使用协议:

```
doorProg : DoorCmd () doorProg = 做响铃
```

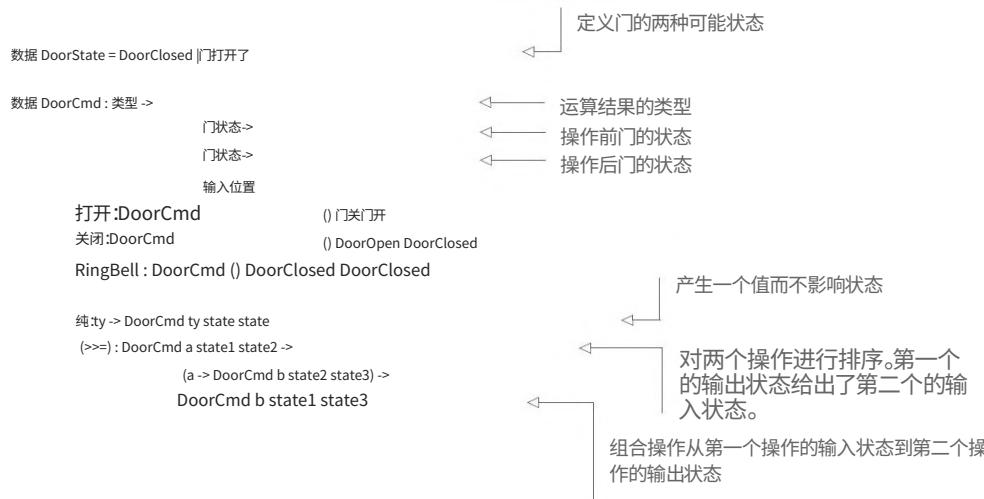
```
    打开  
    关
```

不幸的是,您还可以描述不遵循协议的无效操作序列,例如以下,您尝试打开门两次,然后在门已经打开时按门铃:

```
doorProgBad : DoorCmd() doorProgBad  
= 打开  
    打开  
    响铃
```

您可以通过在DoorCmd操作类型中跟踪门的状态来避免这种情况,并将DoorCmd的功能限制为遵循协议的有效操作序列。以下清单显示了如何执行此操作,准确描述了图 13.2 中在命令类型中表示的状态转换。

**清单 13.2 在一个类型中建模门状态机,描述状态转换
命令的类型 (Door.idr)**



每个命令的类型都接受三个参数:

命令产生的值的类型门的输入状态;即门必须处于的状态才能执行操作门的输出状态;即执行操作后门的状态

因此,以下函数的实现将描述一系列
以关门开始和结束的动作:

```
doorProg : DoorCmd () DoorClosed DoorClosed
```

DOORCMD中的参数顺序 请注意,操作序列产生的类型是DoorCmd 的第一个参数,然后是

输入和输出状态。这是定义类型时的常见约定
用于描述状态转换,它将在第 14 章中变得重要
当我们查看处理错误并从环境反馈的更复杂的状态机时。

一般来说,如果你有一个DoorType ty beforeState afterState 类型的值,它
描述产生类型ty 值的一系列门动作;它始于
门在状态之前的状态;它以状态afterState 中的门结束。

13.1.2 门操作序列的交互开发

要了解DoorCmd中的类型如何帮助您正确编写操作序列,
让我们重新实现doorProg。我们将按照与以前相同的方式编写此代码:按门铃,打开门,然后关闭门。

如果你增量编写,你会看到类型如何显示状态的变化
在整个动作序列中的门:

1 定义 - 从骨架定义开始:

```
doorProg : DoorCmd () DoorClosed DoorClosed
doorProg = ?doorProg_rhs
```

2 优化,输入添加一个动作来敲响门铃:

```
doorProg : DoorCmd () DoorClosed DoorClosed
doorProg = 做响铃
?byProg_rhs
```

如果您现在检查?doorProg_rhs的类型,您会发现它应该是
以DoorClosed状态的门开始和结束的一系列动作:

```
doorProg_rhs : DoorCmd () DoorClosed DoorClosed
```

3 优化,输入接下来,添加一个打开门的动作:

```
doorProg : DoorCmd () DoorClosed DoorClosed
doorProg = 做响铃
    打开
?byProg_rhs
```

如果你现在检查?doorProg_rhs的类型,你会发现它应该以
而是处于DoorOpen状态的门:

```
doorProg_rhs : DoorCmd () DoorOpen DoorClosed
```

4优化失败 如果你现在添加一个额外的打开,门已经在 DoorOpen状态,你会得到一个类型错误:

```
doorProg : DoorCmd() DoorClosed DoorClosed doorProg = do
RingBell Open Open ?doorProg_rhs
```

错误说Open的类型是一个从DoorClosed状态开始的操作,但预期的类型是从DoorOpen状态开始的:

```
门.idr:20:15:
使用预期类型检查 doorProg 的右侧时
DoorCmd () DoorClosed DoorClosed

检查构造函数 Main.>>= 的应用程序时:类型不匹配

DoorCmd () DoorClosed DoorOpen (打开类型)
和
DoorCmd a DoorOpen state2 (预期类型)

具体来说:
类型不匹配
    关门
和
    门打开了
```

5 Refine - 而是通过关闭门来完成定义:

```
doorProg : DoorCmd () DoorClosed DoorClosed doorProg = 做响铃

打开
关
```

在类型中定义前置条件和后置条件doorProg 的类型包括为序列提供前置条件和后置条件的输入和输出状态(门必须在序列之前和之后都关闭)。如果定义违反任何一个,您将收到类型错误。

例如,您可能忘记关门:

```
doorProg : DoorCmd () DoorClosed DoorClosed doorProg = 做响铃
```

打开

在这种情况下,您将收到类型错误:

```
门.idr:18:15:
使用预期类型检查 doorProg 的右侧时
DoorCmd () DoorClosed DoorClosed

检查构造函数 Main.>>= 的应用程序时:类型不匹配
```

DoorCmd () DoorClosed DoorOpen (打开类型)

(继续)

和

DoorCmd () DoorClosed DoorClosed (预期类型)

具体来说:

类型不匹配

门打开了

和

关门

该错误是指最后一步，并表示 Open 从 DoorClosed 移动到 DoorOpen，但预期的类型是从 DoorClosed 移动到 DoorClosed。

通过以这种方式定义 DoorCmd，在类型中明确输入和输出状态，您已经定义了一系列门操作有效的含义。并且通过逐步编写 doorProg，使用一系列步骤和定义其余部分的孔，您可以通过查看孔的类型来查看每个阶段的门状态。

门正好有两种状态，DoorClosed 和 DoorOpen，你可以描述正是当您在门操作类型中将状态从一种更改为另一种时。但并非所有系统都有您可以确定的确切状态数

进步。接下来，我们将了解如何对具有无限数量的系统进行建模可能的状态。

13.1.3 无限状态：自动售货机建模

在本节中，我们将使用类型驱动开发为自动售货机建模，编写明确描述每个操作的输入和输出状态的类型。为简化起见，机器只接受一种类型的硬币（1 英镑硬币）并分发一个产品（巧克力棒）。即便如此，也可能存在任意数量的机器中的硬币或巧克力棒，因此可能状态的数量不是有限的。

表 13.1 描述了自动售货机的基本操作，以及每次操作前后的机器状态。

表 13.1 自动售货机操作，输入和输出状态表示为 Nat

硬币（之前）	巧克力（之前）	手术	硬币（后）	巧克力（后）
磅	冲击	投币	磅	冲击
磅	S 冲击	售卖巧克力	磅	冲击
磅	冲击	归还硬币	从	冲击

与门示例一样，每个操作都有一个前置条件和一个后置条件：

前提条件 必须放入的硬币数量和巧克力数量

操作前的机器

状态机 : 在类型中跟踪状态

后置条件 机器中硬币的数量和巧克力的数量

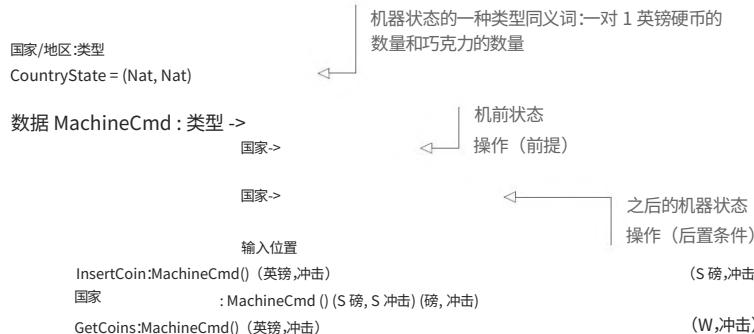
手术后。

您可以将机器的状态表示为一对两个Nat,第一个表示机器中硬币的数量,第二个表示巧克力的数量:

国家/地区:类型
CountryState = (Nat, Nat)

下一个清单将自动售货机状态表示为 Idris 类型,表 13.1 中的状态转换显式写入MachineCmd操作的类型中。

清单 13.3 在类型中建模自动售货机,描述命令类型中的状态转换 (Vending.idr)



要完成模型,您需要能够对命令进行排序。您还需要能够阅读用户输入:您定义的命令描述了机器的功能,但还有一个用户界面,包含以下内容:

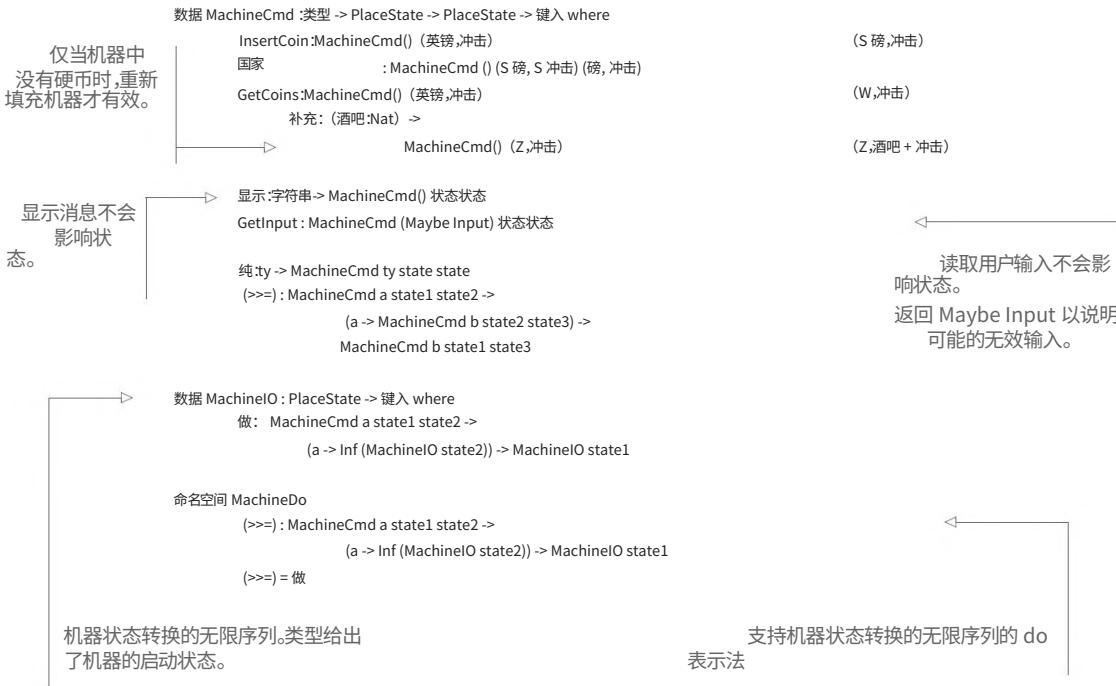
投币口

一个出售按钮,用于分发巧克力一个更改按钮,用于退回任何未使用的硬币

您可以在数据类型中对这些操作进行建模,以描述可能的用户输入。清单 13.4 显示了自动售货机的完整模型,包括显示消息(Display)、重新填充机器(Refill)和读取用户操作(GetInput) 的附加操作。

清单 13.4 自动售货机状态的完整模型 (Vending.idr)





13.1.4 经验证的自动售货机说明

清单 13.5 显示了一个函数的概要,该函数描述了使用MachineCmd定义的状态转换的自动售货机的经过验证的操作序列。只要它进行类型检查,您就知道您已经正确地对操作进行了排序,并且您永远不会再在不满足其先决条件的情况下执行操作。

清单 13.5 读取和处理自动售货机用户输入的主循环
(Vending.idr)



状态机:在类型中跟踪状态

```
改变 => 做 GetCoins
显示 “更改返回” machineLoop
REFILL num => 重新填充 num
```

有出售和补充的孔。在每种情况下,您都需要检查硬币和巧克力的数量是否满足其先决条件。如果你在不检查前置条件的情况下尝试Vend ,Idris 会报错:

```
卖出:MachineIO (英镑,冲击)卖出 = 卖出
显示 “享受！”机器循环
不进行类型检查,因为机器中可能没有硬币或巧克力
```

Idris 会报错,因为你没有检查机器里是否有硬币和巧克力棒,所以可能不满足前提条件:

Vending.idr:67:13:当检查带有预期
类型 MachineIO (磅、巧克力)的自动售货机右侧时

检查函数 Main.MachineDo.>=> 的应用程序时:
类型不匹配
 机器命令 ()
 (S 磅 1,S chocs2) (磅 1,chocs2)
 (Vend 类型)
 和
 MachineCmd () (pounds, chocs) (pounds1, chocs2) (预期类型)

具体来说:
 类型不匹配
 S chocs1
 和
 冲击

该错误表明输入状态必须是(Spounds1, S chocs2)的形式,但它是(pounds, chocs) 的形式。

您可以通过对隐式参数、磅和巧克力进行模式匹配来解决此问题,以确保它们采用正确的形式,否则会显示错误。下面的清单显示了执行此操作的vend和refill的定义。

清单 13.6 添加 vend 和 refill 的定义以检查它们的先决条件是否得到满足 (Vending.idr)

```
卖出:MachineIO (磅,冲击)卖出 {磅 = S p} {冲击 = S c}
= 做 Vend
显示 “享受！” machineLoop
vend {磅 = Z}
    机器里没有钱; can t vend = do Display
Insert a coin machineLoop vend {chocs = Z}
    机器里没有巧克力;不能售卖= 显示 “缺货”
```

机器循环

```

补充: (编号:Nat) -> MachineIO (磅,冲击)
补充 {磅 = Z} 数量
    = 重新填充 num
        机器循环
笔芯      _ = do 显示 "无法重新填充:机器中的硬币"
        机器循环

```

仅当机器中没有硬币时,Refill 才允许用巧克力补充库存。

对于门和自动售货机,我们使用类型来模拟物理系统。在每种情况下,类型都给出了系统所处状态的抽象。在每个操作之前和之后,并且类型中的值描述了有效的序列操作。我们还没有实现运行函数来执行状态转换。对于DoorCmd或MachineCmd,但在本书随附的代码中,即在线提供,您将找到实现自动售货机控制台模拟的代码机器。

在下一节中,您将看到一个更具体的跟踪状态示例类型,实现堆栈数据结构。我将用这个例子来说明你如何可以在实践中执行命令。

练习

1更改RingBell操作,使其在任何状态下工作,而不是仅在那扇门是关着的。您可以通过查看以下函数来测试您的答案
类型检查:

```

doorProg : DoorCmd () DoorClosed DoorClosed
doorProg = 做响铃
    打开
    响铃
    关

```

2以下(不完整)类型定义了一个猜谜游戏的命令,其中输入和输出状态是允许的剩余猜测数:

```

数据 GuessCmd :类型 -> Nat -> Nat -> 键入位置
尝试:整数 -> GuessCmd 排序 ?in_state ?out_state

Pure : ty -> GuessCmd ty state state
(>>=) : GuessCmd a state1 state2 ->
(a -> GuessCmd b state2 state3) ->
GuessCmd b state1 state3

```

Try命令返回一个Ordering,说明猜测是否太高,太低或正确,这会改变可用猜测的数量。完成Try的类型,以便您只能在至少有一个猜测时进行猜测允许,因此猜测减少了可用猜测的数量。

如果您有正确的答案,则应对以下定义进行类型检查:

```

三猜测:GuessCmd() 3 0
threeGuesses = 尝试 10

```

```
试试 20
试试 15
纯的 ()
```

此外,以下定义不应进行类型检查:

```
noGuesses : GuessCmd () 0 0
noGuesses = 尝试 10
纯的 ()
```

3以下类型定义了可能的物质状态:

数据物质 = 固体 | 液体 | 气体

以以下定义类型检查的方式定义MatterCmd类型:

```
iceSteam : MatterCmd () 固体气体
iceSteam = 融化
熬

steamIce : MatterCmd () 气体固体
steamIce = 做冷凝
冻结
```

此外,以下定义不应进行类型检查:

```
overMelt : MatterCmd() 固体气体
overMelt = 做融化
熔化
```

13.2 状态中的依赖类型:实现堆栈

您已经了解了如何为两个抽象示例在类型中建模状态转换:一扇门
(表示它的类型是打开还是关闭)和自动售货机(表示其类型的内容)。将此抽象信息存储在

当您还有与之相关的具体数据时,操作特别有用
抽象数据。例如,如果您要描述特定大小的数据,以及
一个操作告诉你它如何改变数据的大小,你可以使用Vect作为具体的表示。您将知道输入和输出Vect所
需的长度
从每个操作的类型。

在本节中,您将通过在堆栈上实现操作来了解其工作原理
数据结构。堆栈是一种后进先出的数据结构,您可以在其中添加项目
并将它们从堆栈的顶部移除,并且只有顶部的项目可以访问。一个
stack 支持三种操作:

- Push** 将新项目添加到堆栈顶部
- Pop** 从堆栈中移除顶部项目,前提是堆栈不为空
- Top** - 检查堆栈上的顶部项目,前提是堆栈不为空

与自动售货机上的操作一样,这些操作中的每一个都有一个描述必要输入状态的前置条件和一个描述输出状态的后置条件。表 13.2 描述了这些,在每个之前给出了所需的堆栈大小

操作和操作后产生的堆栈大小。

表 13.2 堆栈操作,输入和输出堆栈大小表示为Nat

堆栈大小 (之前)	手术	堆栈大小 (之后)
高度	推动元件	S 高度
S 高度	流行元素	高度
S 高度	检查顶部元素	S 高度

您将在每个操作的类型中表达前置条件和后置条件。
一旦你在堆栈上定义了操作,你将实现一个函数来运行
堆栈操作序列,使用堆栈的具体表示及其
高度在它的类型。因为您在状态转换中使用堆栈的高度,所以
堆栈的良好具体表示是Vect。例如,你知道一个堆栈
高度为10的整数,正好包含 10 个整数,因此您可以将其表示为
Vect 10 整数类型的值。
最后,您将看到一个实际的堆栈示例,它实现了基于堆栈的
交互式计算器。

13.2.1 在状态机中表示堆栈操作

与第 13.1 节中的DoorCmd和MachineCmd一样,我们将描述堆栈上的操作
在依赖类型中并放置输入和输出状态的重要属性
在类型中显式。在这里,我们感兴趣的国家的财产是高度
堆栈。

清单 13.7 展示了如何用代码表达表 13.2 中的操作,
描述每个操作如何影响堆栈的高度。对于此示例,您将
仅在堆栈上存储整数值,但您可以扩展StackCmd以允许
通过参数化堆栈中的元素类型来实现通用堆栈。

清单 13.7 用输入和输出表示堆栈数据结构上的操作
将堆栈的高度放入类型 (Stack.idr)

您将使用 Vect 来表示堆栈,因此
在此处导入 Data.Vect。

导入 Data.Vect

```
数据 StackCmd :类型 -> Nat -> Nat -> 键入位置
Push : Integer -> StackCmd () 高度 (S 高度)
Pop : StackCmd 整数 (S 高度) 高度
顶部:StackCmd 整数 (S 高度) (S 高度)
```

```
纯:ty -> StackCmd ty height 高度
(>>=) : StackCmd a height1 height2 ->
(a -> StackCmd b height2 height3) ->
StackCmd b height1 height3
```

Pop 要求堆栈上至少有一个元素,它将堆栈的
高度减少 1。

Push 将堆栈的高度增加 1。

顶部需要在
上的至少一个元素
堆栈,它保留了
堆栈的高度。

状态中的依赖类型:实现堆栈

您使用Vect来表示堆栈,因此每次向向量添加元素或删除元素时,都会更改向量的类型。因此,您通过将相关参数放入 StateCmd 类型本身类型的 (Vect 的长度) 来表示依赖类型的可变状态。

使用StackCmd,您可以编写堆栈操作序列,其中堆栈的输入和输出高度在类型中是明确的。例如,以下函数压入两个整数,弹出两个整数,然后返回它们的和:

```
testAdd : StackCmd 整数 0 0 testAdd = 做推送 10
```

```
    推 20
    val1 <- 流行 val2
    <- 流行
    纯 (val1 + val2)
```

StackCmd 中构造函数的类型确保当您尝试Pop 时,堆栈上始终有一个元素。例如,如果你在testAdd 中只 push 一个整数, Idris 会报错:

```
testAdd : StackCmd 整数 0 0 testAdd = 做推送 10
```

```
    val1 <- 流行 val2
    <- 流行
    纯 (val1 + val2)
```

堆栈上只有一个元素,因此 Pop 不
进行类型检查。

当您尝试像这样定义testAdd时,Idris 报告错误:

```
Stack.idr:27:22:
在使用预期类型检查 testAdd 的右侧时
    StackCmd 整数 0 0
```

检查构造函数 Main.>>= 的应用程序时,类型不匹配

StackCmd 整数 (S 高度)高度 (弹出类型)
和

StackCmd a 0 height2 (预期类型)

具体来说:

类型不匹配
S 高度
和
0

这个错误,尤其是 S 高度和 0 之间的不匹配,意味着您有一个高度为 0 的堆栈,但 Pop 需要一个包含至少一个元素的堆栈。

这种方法类似于第 12 章中定义的有状态函数,这里使用 Push 和 Pop 来描述如何修改和查询状态。与之前对有状态操作序列的描述一样,您需要编写一个单独的函数来运行这些序列。

13.2.2 使用Vect实现栈

清单 13.8 展示了如何实现一个执行堆栈操作的函数。这类似于您在第 12 章中看到的 runState,但在
这里您将正确高度的输入 Vect 作为堆栈的内容。

**清单 13.8 在堆栈上执行一系列动作,使用Vect来表示
堆栈的内容**

```
runStack : (stk : Vect inHeight Integer) ->
    StackCmd ty inHeight outHeight -> (ty, Vect outHeight
    Integer) runStack stk (Push val) = ((), val :: stk) runStack
    (val :: stk) Pop = (val, stk) runStack (val :: stk)顶部 = (val, val :: stk)

runStack stk (Pure x) = (x, stk) runStack stk (cmd >>=
next)
= let (cmdRes, newStk) = runStack stk cmd in
runStack newStk (下一个 cmdRes)
```

如果你用 testAdd 尝试 runStack ,给它传递一个初始的空栈,你会看到它返回你推送的两个元素的总和,并且最终的栈是空的:

```
*Stack> runStack [] testAdd
(30, []) : (整数, Vect 0 整数)
```

您还可以定义如下函数,将堆栈顶部的两个元素添加到堆栈中,然后将结果放回堆栈:

```
doAdd : StackCmd () (S (S height)) (S height) doAdd = do val1 <- Pop val2 <-
Pop

    推 (val1 + val2)
```

输入状态 S (S 高度)意味着堆栈上必须至少有两个元素,否则,它可以是任何高度。如果您尝试使用包
含两个元素的初始堆栈执行 doAdd ,您会看到它会生成一个包含单个元素的堆栈,该元素是两个输入
元素的总和:

```
*Stack> runStack [2,3] doAdd
((), [5]) : ((), Vect 1 整数)
```

如果输入状态包含两个以上的元素,您会看到它导致堆栈的高度比输入高度小一。例如, [2, 3, 4]的
输入堆栈会产生一个值为[2 + 3, 4] 的输出堆栈:

```
*Stack> runStack [2,3,4] doAdd
((), [5, 4]) : ((), Vect 2 整数)
```

状态中的依赖类型:实现堆栈

您可以通过另一个调用doAdd 将这两个元素添加到结果堆栈中:

```
*Stack> runStack [2,3,4] (做doAdd;doAdd)
(),[9]):(),Vect 1 整数
```

但是再尝试一个doAdd会导致类型错误,因为堆栈上只剩下一个元素:

```
*Stack> runStack [2,3,4] (do doAdd; doAdd; doAdd) (input):1:34:检查构造函数Main的应用时。
```

```
>>=:
```

类型不匹配
StackCmd () (S (S 高度)) (S 高度) (doAdd 的类型)
和
StackCmd ty 1 outHeight (预期类型)

具体来说:

类型不匹配
S 高度
和
0

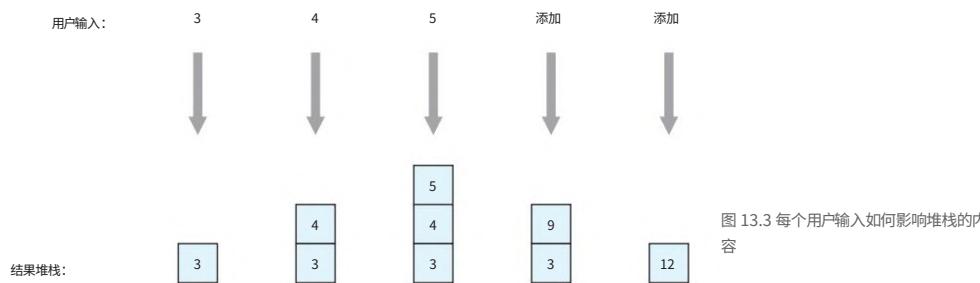
此错误意味着您需要堆栈上的S (S 高度)元素 (即至少两个元素)但您只有S 高度 (即至少一个,但不一定更多)。因此,通过将堆栈的高度放入类型中,您已经明确指定了每个操作的前置条件和后置条件,因此如果您违反其中任何一个,您都会收到类型错误。

13.2.3 交互使用堆栈:基于堆栈的计算器

如果添加从控制台读取和写入的命令,则可以编写控制台应用程序来操作堆栈并实现基于堆栈的计算器。用户可以输入一个数字,将数字压入堆栈,或者添加,将堆栈顶部的两个项目相加,将结果压入堆栈并显示结果。典型的会话可能如下所示:

```
*堆栈IO> 执行
> 3
> 4
> 5      ↗ 用户将三个值压入堆栈:3、4 和 5
          ↙
> 添加    ↗ 添加栈顶两个项,显示并推送结
9          果
> 添加
12
> 添加
堆叠中少于两个项目      ↗ 错误,因为上面只有一项 (12)
```

图 13.3 显示了此会话中的每个有效输入如何影响堆栈的内容。用户每输入一个整数,栈大小就加一,用户每输入一个add,栈大小就减一,只要有两个项目要加。



要实现这个交互式堆栈程序,您需要扩展StackCmd以支持从控制台读取和写入。以下清单显示了新文件StackIO.idr中的StackCmd,该文件扩展了两个命令: GetStr和PutStr。

**清单 13.9 使用 GetStr 命令扩展StackCmd以支持控制台 I/O
和PutStr (StackIO.idr)**

```
数据 StackCmd :类型 -> Nat -> Nat -> 键入位置
Push : Integer -> StackCmd () 高度 (S 高度)
Pop : StackCmd 整数 (S 高度) 高度
顶部 : StackCmd 整数 (S 高度) (S 高度)

GetStr : StackCmd 字符串高度 高度
PutStr : String -> StackCmd () height 高度

纯:ty -> StackCmd ty height 高度
(>>=) : StackCmd a height1 height2 ->
    (a -> StackCmd b height2 height3) ->
    StackCmd b height1 height3
```

GetStr 和 PutStr 都不使用堆栈,
因此高度保持不变。

效果库中的依赖状态 我在第 12 章中提到了效果库,它允许您组合状态和控制台I/O等效果,而无需定义新类型,如此处的StackCmd。效果库支持状态转换和依赖状态的描述,如Stack Cmd 中一样。我不会在本书中进一步描述 Effects 库,但是在这里学习依赖状态的原理意味着你将能够更容易地学习如何使用更灵活的 Effects 库。

您还需要更新runStack以支持这两个新命令。因为Get Str和PutStr描述了交互操作,所以您需要更新run Stack的类型以返回IO操作。这是更新的runStack。

清单 13.10 更新runStack以支持交互式命令GetStr和PutStr (StackIO.idr)

```
runStack : (stk : Vect inHeight Integer) ->
    StackCmd ty inHeight outHeight ->
    IO (ty, Vect outHeight Integer) runStack stk (Push
val) = pure ((), val :: stk)
```

状态中的依赖类型:实现堆栈

```

runStack (val :: stk) Pop = pure (val, stk) runStack (val :: stk) Top = pure (val, val ::  

stk) runStack stk GetStr = do x <- getLine  

    纯 (x,stk)  

runStack stk (PutStr x) = 做 putStr x  

    纯 ((), stk)  

runStack stk (Pure x) = pure (x, stk) runStack stk (x >>= f) = do (x ,  

newStk) <- runStack stk x  

    runStack newStk (f x )

```

与自动售货机一样,您将通过定义一个单独的StackIO类型来描述堆栈操作的无限流,从而在总函数中描述无限序列的StackCmd操作。下面的清单显示了在给定堆栈初始状态的情况下,如何定义StackIO以及如何运行StackIO序列。

清单 13.11 定义交互式堆栈操作的无限序列 (StackIO.idr)

```

数据 StackIO : Nat -> 在哪里输入
    做: StackCmd a height1 height2 -> (a -> Inf (StackIO height2))
        -> StackIO height1

命名空间 StackDo
    (>>=) : StackCmd a height1 height2 ->
        (a -> Inf (StackIO height2)) -> StackIO height1
    (>>=) = 做

数据燃料 = 干燥 | 更多 (懒惰的燃料)
部分永远:燃料
永远=永远更多

run : Fuel -> Vect height Integer -> StackIO height -> IO () run (More fuel) stk (Do cf)
    = do (res, newStk) <- runStack stk c
        运行燃料 newStk (f res)
    运行 Dry stk p = pure ()

```

Nat 参数是无限序列的堆栈的初始高度。

支持 do 表达法堆栈IO

永远允许您通过提供无限的燃料来无限期地运行整个程序。详情请参阅第 11 章。

输入 Vect 必须具有由初始堆栈高度给出的多个项目。

交互式计算器遵循与自动售货机实现类似的模式。下一个清单显示了主循环的概要,它读取输入,将其解析为命令类型,并在输入有效时处理命令。

清单 13.12 一个基于堆栈的交互式计算器的概要 (StackIO.idr)

```

数据 StkInput = 数字整数
    添加
strToIntput : String -> 也许是 StkInput
    相互的
tryAdd : StackIO 高度
stackCalc : StackIO 高度 stackCalc = do PutStr
    >
    输入 <- GetStr

```

描述可能的用户输入:输入数字或添加

解析从控制台读取的输入。
返回可能是因为输入可能无效。

在堆栈顶部添加两个数字 (如果存在),然后循环

交互式计算器的主循环

```

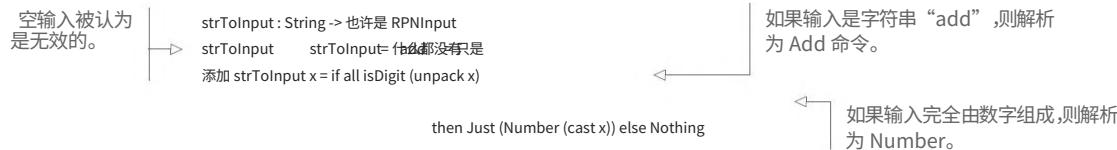
case strToInput 输入
    什么都没有 => 做 PutStr 无效输入\n
        堆栈计算器
    Just (Number x) => 做 Push x
        堆栈计算器
    只需添加 => tryAdd

main : IO () main = 永
远运行 [] stackCalc

```

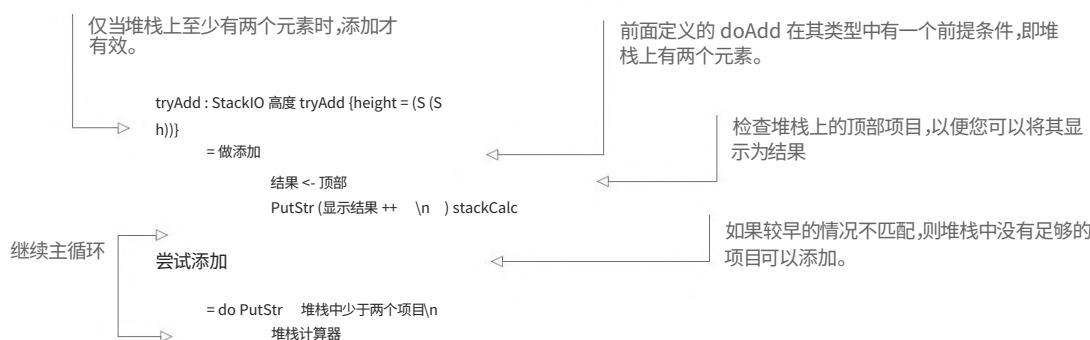
如果可能,您仍然需要定义解析用户输入的strToInput和将两个元素添加到堆栈顶部的tryAdd。以下清单显示了strToInput 的定义。

清单 13.13 读取基于堆栈的计算器的用户输入 (StackIO.idr)



最后,下一个清单显示了tryAdd 的定义。就像自动售货机实现中的vend和refill一样,您需要匹配初始状态以确保堆栈上有足够的项目可以添加。

清单 13.14 添加栈顶的两个元素,如果它们存在的话 (StackIO.idr)



您可以在REPL中检查stackCalc是否为总计:

```
*StackIO> :total stackCalc Main.stackCalc 是
Total
```

通过将循环组件 (StackIO)与终止组件分离 (StackCmd),并通过为操作提供精确的类型,您可以确保堆栈 Calc至少具有以下属性,只要它是总计的:

它将无限期地继续运行。 它永远不会因为未处理的用户输入而崩溃。 它永远不会因为堆栈溢出而崩溃。

练习



- 1** 将用户命令添加到基于堆栈的计算器中以进行减法和乘法运算。你可以按如下方式测试这些：

```
*ex_13_2> :执行
>5
>3
>减去
2
>8
>相乘
16
```

- 2** 向基于堆栈的计算器添加一个否定用户命令，用于对堆栈上的顶部项目求反。您可以按如下方式进行测试：

```
>10
>否定
-10
```

- 3** 添加一个丢弃用户命令，从堆栈中删除顶部项目。您可以按如下方式进行测试：

```
>3
>4
>丢弃
丢弃 4
>添加
堆叠中少于两个项目
```

- 4** 添加复制堆栈顶部项目的复制用户命令。您可以按如下方式进行测试：

```
>2
>重复
复制 2 > 添加
4
```

13.3 总结

数据类型可以通过使用每个数据构造函数来描述状态转换来对状态机进行建模。

您可以通过将系统的输入和输出状态作为命令类型的一部分来描述命令如何改变系统状态。 使用孔以交互方式开发状态转换序列，意味着您可以检查命令序列所需的输入和输出状态。

372

第13章状态机:验证类型中的协议

类型可以模拟无限状态空间以及有限状态。命令序列给出经过验证的状态转换序列,因为命令序列只有在描述有效的状态转换序列时才会进行类型检查。

您可以通过将依赖类型的参数放入状态转换中来表示可变的依赖类型状态。例如,您可以使用向量的长度来表示堆栈的高度。

14 依赖状态机:处理反馈和 错误

本章涵盖

处理状态转换中的错误以交互方式开发协议实现保证类型中的安全属性

正如您在前一章中看到的,您可以在依赖类型中描述状态机的有效状态转换,由操作所需的输入状态(其前置条件)和输出状态(其后置条件)索引。通过在类型中定义有效的状态转换,您可以确保保证类型检查的程序能够描述有效的状态转换序列。

您看到了两个示例,一个门的描述和一个自动售货机的模拟,并且在每种情况下,我们都为操作提供了精确的类型,以描述它们如何影响状态。但是我们没有考虑任何操作失败的可能性:

如果当您尝试打开门时,门被卡住了怎么办?如果,即使您运行了Open操作,它仍然处于DoorClosed状态怎么办?

如果您在自动售货机中插入硬币时,机器拒绝了您的硬币怎么办?
硬币?

在几乎任何现实环境中,当您尝试给出精确的类型来描述状态机时,您需要考虑操作失败或意外响应的可能性:与DoorState一样,您可能表示类型中的文件句柄(打开或关闭),但如果尝试打开文件,则该文件可能不存在或您可能无权打开它。您可以在状态机中表示一个安全通信协议,但是您能否在该协议中取得进展取决于从网络接收到对您发送的任何请求的有效响应。

你可以用一种类型来表示银行的自动柜员机(ATM)的状态(等待卡、等待PIN输入等),但你只能移动到机器可以在检查时提取现金的状态用户的PIN成功。

在这些情况下,您无法完全控制系统状态的变化方式。您可以请求更改状态(打开文件、发送消息等),但在实践中状态是否更改以及如何更改取决于您从环境收到的响应。在本章中,您将看到如何通过允许状态转换依赖于操作的结果来处理操作失败的可能性。

您还将看到如何处理可能依赖于用户输入的状态变化。我们将研究模拟ATM所涉及的状态转换,其中用户输入决定了机器是否可以分发现金。我们将从第13章中的门模型开始,看看如何处理门可能会卡住的可能性,因此“打开”操作失败。

14.1 处理状态转换中的错误 在上一章中,您定义了一个DoorCmd

数据类型,用于对门的状态转换进行建模,如图14.1中的状态转换图所示。

清单14.1概括了DoorCmd的定义,它模拟了图14.1中的状态转换系统。

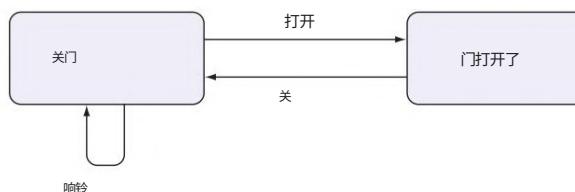


图14.1 显示门上的状态和操作的状态转换图

清单 14.1 在类型中建模门状态机

```
数据 DoorState = DoorClosed ||| 门关闭了
数据 DoorCmd :类型 -> DoorState -> DoorState -> 键入 where
    打开:DoorCmd          () 门关门开
    关闭:DoorCmd          () DoorOpen DoorClosed
    RingBell :DoorCmd () DoorClosed DoorClosed

纯:ty -> DoorCmd ty state state
(>>=) :DoorCmd a state1 state2 -> (a -> DoorCmd b state2
state3) ->
    DoorCmd b state1 state3
```

在此模型中,您可以完全控制每个操作如何从一种状态移动到另一种状态。例如, Open的类型声明它总是以处于DoorClosed状态的门开始,并且总是以处于DoorOpen状态的门结束:

打开 :DoorCmd () DoorClosed DoorOpen

然而,现实并不总是那么包容!如果你用一些真实的硬件来实现这个,比如按下按钮操作的滑动门,你需要考虑硬件问题的可能性,比如门卡。在本节中,我们将改进行模型以捕捉这种失败的可能性,并了解这如何影响遵循协议的程序的实现。

14.1.1 完善门模型:代表失败

打开可能会因为门被卡住而失败,所以我们需要一种方法来表示它是否成功。我们可以定义一个枚举类型来描述开门的可能结果:

数据门结果 = OK |卡住

然后, Open可以返回一个DoorResult,而不是生成单位值。我们可以尝试以下类型的Open:

打开 :DoorCmd DoorResult DoorClosed DoorOpen

不幸的是,这并不完全正确,因为它仍然说打开门会导致门处于DoorOpen状态,无论发生什么。不知何故,我们需要表达Open导致门处于DoorClosed或DoorOpen状态,这取决于它产生的DoorResult的值。图14.2展示了我们想要实现的状态机。

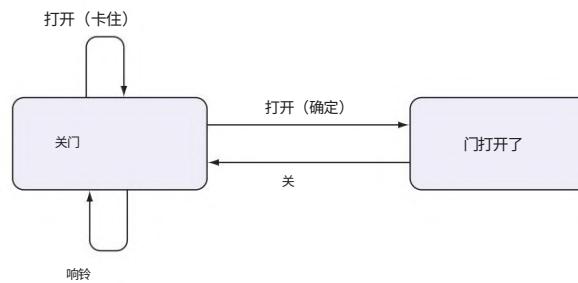


图 14.2 状态转换图，显示门上的状态和操作，其中开门可能失败

您可以通过更改DoorCmd的类型以允许从返回值计算输出状态来实现此目的。图 14.3 说明了如何优化 DoorCmd 的类型来实现这一点。

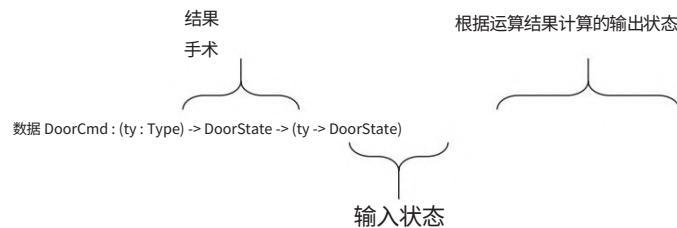


图 14.3 DoorCmd 的新类型，其中操作的输出状态是根据操作的返回值计算的

在这里，您为操作的返回类型ty 命名，并表示输出状态是由一个以ty作为输入的函数计算的。现在，当您定义 Open 的类型（实际上是所有DoorCmd操作）时，您给出了输出状态的表达式，解释了如何从DoorResult 类型的返回值计算输出状态：

```

打开 : DoorCmd DoorResult DoorClosed (\res => case res
of
  OK => 开门
  卡住 => 关门)
  
```

这准确地编码了图 14.2 中的状态转换图所示的内容。也就是说，Open 的输出状态可以是以下之一：

DoorOpen 如果 Open 返回 OK
 DoorClosed 如果 Open 返回 Jammed

尽管在运行操作之前您不会知道res的值，但您至少可以使用类型来解释在给定Open结果的情况下门可能处于的状态。

清单 14.2 显示了经过改进后的完整DoorCmd类型声明。它还添加了显示，因此您可以在必要时显示日志消息。

**清单 14.2 精化的DoorCmd类型,允许每个操作的输出状态
根据操作的返回值计算 (DoorJam.idr)**

根据 Open 的返回值计算输出状态

您使用 const 表示输出状态不依赖于返回值。

```
数据 DoorCmd : (ty : Type) -> DoorState -> (ty -> DoorState) -> Type where
    打开 : DoorCmd DoorResult DoorClosed (\res => case res of
```

OK => 开门
卡住 => 关门)

```
关闭 : DoorCmd () DoorOpen (const DoorClosed)
RingBell : DoorCmd () DoorClosed ( const DoorClosed )
```

显示 : 字符串 -> DoorCmd () 状态 (常量状态)

```
纯 : (res : ty) -> DoorCmd ty (state_fn res) state_fn
(>>=) : DoorCmd a state1 state2_fn ->
    ((res : a) -> DoorCmd b (state2_fn res) state3_fn) ->
        DoorCmd b state1 state3_fn
```

Pure 的这种类型意味着值 res 可用于计算一系列操作的输出状态。

(>>=) 运算符需要根据第一个操作的输出计算中间状态。

用 const 计算输出状态

这是 const 的类型,在 Prelude 中定义:

```
*DoorJam> :t 常量
常量 : a -> b -> a
```

它忽略它的第二个参数并返回它的第一个参数。所以,如果说一个操作的输出状态为 const Door Closed,这会给你一个忽略函数结果并总是返回 DoorClosed 的函数。

在前面的DoorCmd 定义中,在清单 14.1 中,您使用了(>>=)运算符来解释第一个操作的输出状态应该是第二个操作的输入状态:

```
(>>=) : DoorCmd a state1 state2 -> (a -> DoorCmd b state2 state3) ->
    DoorCmd b state1 state3
```

现在稍微复杂一些,因为第一个操作的返回值会影响第二个操作的输入状态:

```
(>>=) : DoorCmd a state1 state2_fn ->
    ((res : a) -> DoorCmd b (state2_fn res) state3_fn) ->
        DoorCmd b state1 state3_fn
```

这工作如下：

- 1 第一个操作返回 a 类型的值，一旦您知道操作的结果，就会从 state2_fn 函数计算输出状态。
- 2 进行第二次操作时，会知道第一次操作的结果，命名为 res，所以它的输入状态为 state2_fn res。
- 3 组合操作具有 state1 的输入状态（第一个操作的输入状态）和使用 state3_fn 从第二个操作的结果计算的输出状态。

以这种方式定义 DoorCmd 可以让您更精确地定义状态转换，这意味着当您使用 DoorCmd 定义函数时，操作的类型需要您在继续之前执行任何必要的检查。例如，在您尝试打开一扇门后，您无法执行任何进一步的门操作，直到您检查了结果。我们将通过重新访问我们之前的示例 doorProg 来了解它是如何工作的。

14.1.2 一个经过验证的、错误检查的、门协议描述

在第 13 章中，您实现了一个函数，使用 DoorCmd 作为一系列动作来响铃、打开和关闭门，并使用类型来验证动作序列是否有效。你写的 doorProg 如下：

```
doorProg : DoorCmd () DoorClosed DoorClosed doorProg = 做响铃
```

```
    打开
    关
```

现在您已经优化了 DoorCmd 的类型，以便根据操作结果计算输出状态，您需要以不同的方式编写类型：

```
doorProg : DoorCmd () DoorClosed (const DoorClosed)
```

也就是说，输出状态不受结果的影响，因此您使用 const，它忽略它的第二个参数并返回它的第一个参数不变。但是如果你像以前一样尝试实现 doorProg，而不检查 Open 的结果，你会得到一个错误：

```
doorProg : DoorCmd () DoorClosed (const DoorClosed) doorProg = 做响铃
```

```
    打开
    关
```

当您尝试使用 Close 时会发生错误。它的类型要求输入状态是 DoorOpen，但实际上它的输入状态是根据 Open 的结果计算出来的：

检查构造函数 Main.>>= 的应用程序时：类型不匹配

```
DoorCmd () DoorOpen (const DoorClosed) (关闭类型)
和
DoorCmd ()
((\res =>
```

```

案例资源
OK => 开门
卡住 => 关门) _)
(\value => DoorClosed) (预期类型)

```

要查看如何避免此问题,您可以交互开发doorProg ,开始
从以下几点:

```

doorProg : DoorCmd () DoorClosed (const DoorClosed)
doorProg = 做响铃
    打开
    ?byProg_rhs

```

使用孔调试类型错误如果您遇到难以解决的类型错误
自行理解,通常替换掉有问题的部分是一个好主意
有一个洞的程序 (就像我们用?door_Prog_rhs替换Close所做的那样) ,看看预期的类型是什么,以及其
中的任何局部变量
范围。

1类型,细化如果您检查?doorProg_rhs 的类型,您将看到:

```

-----
doorProg_rhs : DoorCmd()
    (案子 _ 的
        OK => 开门
        卡住 => 关门)
    (\value => 关门)

```

您在此处看到的输出状态来自 Prelude 中const的定义,它是一个忽略其参数值并返回Door Closed 的函数。
您要查找的输入状态是根据某个值计算得出的,该值是您尚未命名的Open的结果。让我们称之为果酱:

—°

```

doorProg : DoorCmd () DoorClosed (const DoorClosed)
doorProg = 做响铃
    果酱 <- 打开
    ?byProg_rhs

```

2类型 您现在将看到计算下一个操作的输入状态
从果酱的价值:

```

果酱:按结果
-----
doorProg_rhs : DoorCmd()
    (案例果酱的
        OK => 开门
        卡住 => 关门)
    (\value => 关门)

```

3定义,类型 因为输入状态取决于jam的值,所以可以使
通过检查jam的值来取得进展:

```

doorProg : DoorCmd () DoorClosed (const DoorClosed)
doorProg = 做响铃
    果酱 <- 打开

```

```
case_val => ?
doorProg_rhs 的案例堵塞
```

您现在将看到输入状态取决于 case_val 的值：

```
case_val : DoorResult jam :
DoorResult
-----
doorProg_rhs : DoorCmd()
(case case_val 的
OK => 开门
卡住 => 关门)
(\value => 关门)
```

4 定义,类型 如果对 case_val 进行 case-split ,您应该看到 case 的每个分支都有不同的类型,根据 case_val 的特定值计算得出：

```
doorProg : DoorCmd() DoorClosed(const DoorClosed) doorProg = 做 RingBell jam
<- Open case jam or OK => ?doorProg_rhs_1 Jammed => ?doorProg_rhs_2
```

例如,在?doorProg_rhs_1中, jam的值为OK,因此门必须已成功打开：

```
果酱:按结果
-----
doorProg_rhs_1 : DoorCmd () DoorOpen (\value => DoorClosed)
```

另一方面,在?doorProg_rhs_2 中,门被卡住了,所以它仍然在关门状态:

```
果酱:按结果
-----
doorProg_rhs_2 : DoorCmd () DoorClosed (\value => DoorClosed)
```

5 优化 - 要完成定义,您可以在每种情况下显示一条日志消息,如果门打开,请再次关闭它：

```
doorProg : DoorCmd () DoorClosed (const DoorClosed) doorProg = do RingBell jam
<- Open case jam of
```

```
OK => 显示 “很高兴为您服务”
关
卡住 => 显示 “门卡住”
```

Open类型意味着您需要在执行任何需要了解门状态的进一步操作之前检查门的状态。特别是,除非您成功打开门,否则您无法关闭门。不过,您不必立即检查。例如,您可以在开门和检查结果之间显示一条消息：

```
doorProg : DoorCmd () DoorClosed (const DoorClosed) doorProg = 做响铃堵塞 <- 打开
```

显示“试图开门”的情况下卡纸

```
OK => 显示“很高兴为您服务”
关
卡住 => 显示“门卡住”
```

这是有效的,因为Display的前提条件并不要求门处于特定状态; any 都可以, Display不会改变状态。

使用您在第 5 章中首次看到的模式匹配绑定,您还可以更简洁地定义doorProg ,如下所示:

```
doorProg : DoorCmd () DoorClosed (const DoorClosed) doorProg = 做响铃
```

```
好的 <- 打开 | 卡住 => 显示“门卡住”
显示“很高兴为您服务”
关
```

这提供了通过操作序列的默认路径,当Open返回OK 时,以及当Open返回Jammed时的替代操作。使用模式匹配绑定可以更轻松地编写更长的操作序列,其中一些操作可能会失败。例如,您可以打开和关闭门两次,如果其中一个失败,则放弃该顺序:

```
doorProg : DoorCmd () DoorClosed (const DoorClosed) doorProg = 做响铃
```

```
好的 <- 打开 | 卡住 => 显示“门卡住”
显示“很高兴为您服务”
关
好的 <- 打开 | 卡住 => 显示“门卡住”
显示“很高兴为您服务”
关
```

这个例子在类型中描述了一个协议,它明确地说明了操作可能失败的地方。在doorProg 中, Open类型意味着您需要先检查其结果,然后才能继续执行任何更改状态的进一步操作。

纯的类型

DoorCmd 中 Pure 的类型允许您定义如下函数,其中对 Pure 的调用会更改状态:

```
logOpen : DoorCmd DoorResult DoorClosed (\res => case res of
```

```
OK => 开门
卡住 => 关门)
```

```
logOpen = do Display 试图开门
好的 <- 打开 | 卡住 => 显示“卡住”
纯堵塞
显示“成功”
纯好的
```

(继续)

如果将最后一行 Pure OK 替换为一个洞 ?pure_ok,您会看到它的输入状态为 DoorOpen,而输出状态 (整个 logOpen 函数的)需要是一个计算其输出的函数状态:

```
pure_ok : DoorCmd DoorResult DoorOpen (\res => case
    res of
        OK => 开门
        卡住 => 关门)
```

Pure 的类型旨在在这种情况下工作:

纯 : (res : ty) -> DoorCmd ty (state_fn res) state_fn

这里,state_fn 是包含 case 块的函数,Pure 必须将 OK 作为参数,才能为 ?pure_ok 提供正确的输入状态。

你现在有了一个DoorCmd的定义,它精确地描述了打开和关闭门的协议,捕获了失败的可能性。但是您还没有看到相应的run 函数是如何工作的,这就是在实践中会产生Open操作结果的地方。接下来,我们将在一个更大的例子的背景下来看这个: ATM。我们还将研究状态如何根据用户输入而改变。

14.2 类型中的安全属性:为 ATM 建模

您可以在操作类型中使用显式状态,以通过类型检查来保证系统仅在其处于有效状态时才执行安全关键操作。例如, ATM应该只在用户插入他们的卡并输入正确的 PIN 时才可以提取现金。这是ATM 上的典型操作顺序:

- 1 用户插入他们的银行卡。
- 2 机器提示用户输入 PIN,以检查用户是否有权 使用该卡。
- 3 如果PIN输入成功,机器会提示用户输入一定数量的钱,然后分发现金。

如果用户输入正确的密码,机器将处于出钞状态;否则,它不会。在本节中,我们将为ATM定义一个模型,并了解如何根据用户输入更改机器的状态,在其类型中。

ATM 的安全属性在这个模型中,我们将省略一些银行业务的细节,例如访问和更新用户的银行账户,以及安全地 检查PIN ,这将通过单独的状态机完成。

我们将专注于我们想要维护的一个重要安全属性:机器必须仅在机器中有经过验证的卡时才可以分发现金。

与门模型一样,我们将首先定义 ATM 的可能状态以及可以更改ATM状态的操作。一旦我们知道如何操作

类型中的安全属性 :为 ATM 建模

影响状态,我们可以定义一个数据类型来表示机器上的操作。

14.2.1 为 ATM 定义状态

ATM要么等待用户开始交互,等待用户输入他们的PIN,要么在验证 PIN 后准备分发现金。因此,在我们的模型中,ATM可以处于以下状态之一:

准备就绪 ATM已准备就绪,正在等待插入卡。 CardInserted - ATM 中有一张卡,但系统尚未检查卡上的PIN输入。 会话 ATM中有一张卡,用户输入了有效的PIN

对于卡,因此正在进行验证的会话。

我们将通过检查用户输入正确的PIN 来验证卡。在第 13 章的自动售货机中,您还必须检查输入是否有效,但在这种情况下,您可以在本地检查命令。在这里,我们假设有一个外部服务来检查PIN,所以直到运行时我们才会知道哪些输入会导致哪些状态。

机器支持以下基本操作,每一个都可能对机器的状态有前置条件和后置条件:

InsertCard 等待用户插入卡片

EjectCard从机器中弹出一张卡片,只要卡片中有一张卡片。
机器

GetPIN读取用户的PIN,只要机器中有卡 CheckPIN检查输入的PIN是否有效

Dispense 只要机器中有经过验证的卡,就可以分发现金 GetAmount 从用户那里读取要分发的金额

Message 向用户显示消息

图 14.4 说明了这些操作如何影响机器的状态。

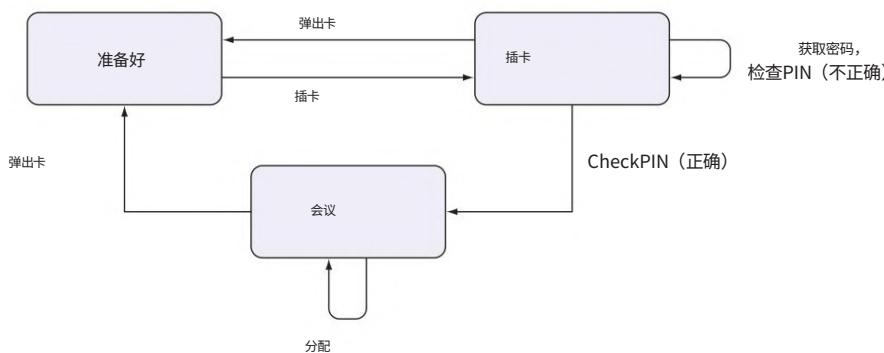


图 14.4 描述 ATM 上的状态和操作的状态机。这省略了在所有状态下都有效的操作 (例如GetAmount) 。

在定义了状态并了解了机器执行的高级操作如何影响状态之后，我们现在可以为ATM定义一种类型，描述图 14.4 所示的状态转换。

14.2.2 定义 ATM 的类型

清单 14.3 定义了一个ATMCmd类型，它表示Idris 代码中ATM上操作的状态转换。它还包括GetAmount 和 Message，它们在所有状态下都有效并且不影响状态，以及通常的操作Pure 和($\gg=$)。EjectCard 的类型略有简化；我们将在 14.2.4 节中对此进行细化。

清单 14.3 表示 ATM 命令的类型，以及它们如何影响 ATM 的状态 (ATM.idr)



使用ATMCmd，您可以编写一个函数来描述ATM 上的会话，从用户插入卡到机器分发现金。清单 14.4 显示了 atm 函数的开始，它等待用户插入卡，提示输入 PIN，然后检查结果。我为序列的其余部分留下了一个洞，我们将其中检查 PIN 并在 PIN 有效时分发现金。

类型中的安全属性 :为 ATM 建模

清单 14.4 描述 ATM 上一系列操作的 atm 函数 (ATM.idr)

```
atm : ATMCmd () 准备就绪 (const Ready) atm = do InsertCard
    pin <- GetPIN pinOK <-
        CheckPIN pin ?atm_rhs
```

检查 PIN 是否有效。
机器的下一个状态取决于 pinOK 的值。

您可以按如下方式完成 atm :

- 1** 类型,定义 如果你检查?atm_rhs孔的类型,你会看到你开始的状态取决于pinOK 的值,你需要结束在Ready状态:

```
pin : Vect 4 Char pinOK : PINCheck
-----
atm_rhs:ATMCmd ()
  (案例pinOK的
    正确PIN => 会话
    不正确的PIN => CardInserted)
  (\value => 准备好了)
```

这里的类型建议您可以通过检查pinOK 的值来继续:

```
atm : ATMCmd () 准备就绪 (const Ready) atm = do InsertCard
    pin <- GetPIN pinOK <-
        CheckPIN pin case pinOK of
            正确PIN => ?atm_rhs_1 不正确PIN => ?
                atm_rhs_2
```

- 2** 类型如果您检查?atm_rhs_1和?atm_rhs_2,您会发现每种情况下的状态都不同。在?atm_rhs_1 中,发现PIN是有效的,因此您有一个经过验证的会话:

```
pinOK : PINCheck pin : Vect 4
Char
-----
atm_rhs_1 : ATMCmd () 会话 (\value => 就绪)
```

另一方面,在?atm_rhs_2 中,发现PIN无效,因此您仍处于CardInserted状态:

```
pinOK : PINCheck pin : Vect 4
Char
-----
atm_rhs_2 : ATMCmd () CardInserted (\value => Ready)
```

- 3 Refine** 在?atm_rhs_1 中,您现在可以分配现金,因此您可以提示输入量和分配:

```
案例pinOK的
CorrectPIN => 做现金 <- GetAmount
          发放现金
```

?atm_rhs_1 不
正确的PIN => ?atm_rhs_2

4 输入,细化 ?atm_rhs_1 的输入状态仍然是Session,所以在完成之前您需要回到Ready状态:

```
pin : Vect 4 Char pinOK :  
PINCheck cash : Nat
```

atm_rhs_1 : ATMCmd () 会话 (\value => 就绪)

您可以通过弹出卡来实现此目的:

```
案例pinOK的  
CorrectPIN => 做现金 <- GetAmount  
分配现金 EjectCard  
IncorrectPIN => ?  
atm_rhs_2
```

5 Refine 对于?atm_rhs_2, PIN无效,所以最简单的做法是弹出
卡片,导致以下完整定义:

```
atm : ATMCmd () 准备就绪 (const Ready) atm = do InsertCard  
  
pin <- GetPIN pinOK <-  
CheckPIN pin case pinOK of  
  
CorrectPIN => 做现金 <- GetAmount  
发放现金  
弹出卡  
不正确的PIN => EjectCard
```

还有其他方法可以定义atm。例如,向用户显示消息会很有帮助。此外,在实践中, ATM通常在分发现金之
前才检查PIN。下一个清单显示了这种实现atm的替代方法。

清单 14.5 atm的另一种实现,包括给用户的的消息和稍后检查 PIN (ATM.idr)

```
atm : ATMCmd () 准备就绪 (const Ready) atm = do InsertCard
```

```
pin <- GetPIN cash <-  
GetAmount  
pinOK <- CheckPIN 密码  
消息 “Checking Card”案例pinOK
```

您尚未检查 PIN 或 pinOK 的状态,但这些命
令是有效的,因为它们不需要机器处于特定状
态。

```
正确PIN => 分配现金  
弹出卡  
消息 “请取出您的卡和现金”  
IncorrectPIN => 执行消息 “Incorrect PIN”  
弹出卡
```

类型中的安全属性 :为 ATM 建模

只要您只在机器处于适当状态时执行操作,并且只要您确保通过atm中操作的每条路径都以就绪状态结束,您就可以随心所欲地实现细节。如果自动柜员机进行类型检查,您可以确定您已经维护了重要的安全属性:机器只有在机器中有卡并且输入正确的PIN时才会分发现金。

14.2.3 在控制台模拟ATM:执行ATMCmd

要尝试atm功能,您可以编写一个 ATM 的控制台模拟,该ATM生成 ATMCmd描述的IO操作:

```
runATM:ATMCmd res inState outState_fn -> IO res
```

给定一个产生res类型结果的ATMCmd序列,从状态inState 开始,并使用 outState_fn 计算结果状态, runATM给出一个产生结果res的IO操作序列。让我们为此模拟硬编码一个有效的PIN :

```
testPIN : Vect 4 字符
testPIN = [ 1 , 2 , 3 , 4 ]
```

以下清单显示了使用此硬编码PIN的ATM控制台模拟。在这个模拟中,许多命令在控制台上提示输入并返回读取的值。

清单 14.6 ATM 的控制台模拟 (ATM.idr)

```
readVect : (n : Nat) -> IO (Vect n Char) readVect Z = do discard <- getLine
pure [] readVect (S k) = do ch <- getChar
                            chs <- readVect k
                            pure (ch :: chs)

runATM : ATMCmd res inState outState_fn -> IO res runATM InsertCard = do putStrLn "请输入您的卡 (按回车键)" x <- getLine pure ()
runATM EjectCard = putStrLn "卡弹出" runATM GetPIN = do putStrLn "输入 PIN:" readVect 4
runATM (CheckPIN pin) = if pin == testPIN
                           then pure CorrectPIN
                           else pure IncorrectPIN
runATM GetAmount = do putStrLn "你想要多少钱?" x <- getLine
                      pure (cast x) runATM (Dispense amount)
                      = putStrLn ("这里是" ++ show amount)

runATM (Message msg) = putStrLn msg runATM (Pure res) = pure
res runATM (x >= f) = do x <- runATM x
                           putStrLn ("运行ATM (f x )")
```

您现在已经将ATM定义为一种类型，其中的命令描述了ATM上的每个状态转换，以及一个单独的runATM函数来解释IO上下文中的这些命令。通过将描述与实现分开，您可以根据需要为不同的上下文编写不同的解释器。特别是，您不希望在真实设备上硬编码PIN！

14.2.4 使用自动隐式优化前置条件

图 14.4 中ATMCmd类型不能完全捕获的状态机的一个特征是，只有当机器中有卡时才允许弹出卡。

相反，您有以下类型：

```
EjectCard : ATMCmd () 状态 (常量就绪)
```

也就是说，您可以尝试在任何输入状态下弹出卡片，即使机器中没有卡片。但是只有两种状态可以弹出卡片：`CardInserted`和`Session`。您应该无法编写以下函数，因为机器正在弹出处于就绪状态的卡：

```
badATM : ATMCmd () 准备就绪 (const Ready) badATM = EjectCard
```

不知何故，您需要以下两种类型才能为EjectCard 工作：

```
EjectCard : ATMCmd () CardInserted (const Ready)
EjectCard : ATMCmd () 会话 (常量就绪)
```

像EjectCard这样的数据构造函数不能有两种不同的类型。但是，您可以在ATMState上定义一个谓词，该谓词允许您将 EjectCard 的可能输入状态限制为有效的输入状态。我们在第 9 章讨论了谓词，您可以定义一个HasCard谓词来描述机器包含卡片的状态：

```
数据 HasCard : ATMState -> 假设 where
    这是          : HasCard CardInserted
    HasSession : HasCard 会话
```

当state是Card之一时，您只能构造HasCard state类型的值
`Inserted`或`Session`，因此您可以细化EjectCard的类型，如下所示：

```
EjectCard : HasCard 状态 -> ATMCmd () 状态 (常量就绪)
```

如果你这样做，你需要在使用Eject Card时明确地给出HasCard类型的值。例如：

```
insertEject : ATMCmd () 就绪 (常量就绪) insertEject = do InsertCard
```

```
    EjectCard HasCI
```

必须为谓词编写显式值将很快变得乏味。相反，您可以对谓词使用自动隐式，您在第 9 章中也看到了这一点：

```
EjectCard : {auto prf : HasCard state} -> ATMCmd () 状态 (常量就绪)
```

现在,您可以像以前一样使用EjectCard ,并让 Idris 通过搜索HasCard的可能数据构造函数来查找谓词的正确值,以查看它们中的任何一个是否有效:

```
insertEject : ATCMcmd () 就绪 (常量就绪) insertEject = do InsertCard
```

弹出卡

对于badATM, Idris 应该找不到合适的值:

```
badATM : ATCMcmd () 准备就绪 (const Ready) badATM = EjectCard
```

这种情况下,Idris会报错,说需要找到一个HasCard类型的值
准备好EjectCard的谓词,但找不到:

检查构造函数 Main.EjectCard 的参数 prf 时:找不到类型的值

有卡就绪

您之前的所有其他定义,包括两个版本的atm和执行函数runATM,都将使用这个改进的EjectCard 版本而无需任何更改即可工作。

练习



1以下类型概述了用户可以使用通行证登录的安全系统

word ,然后阅读了一个秘密信息 ,虽然有一些差距:

```
数据访问 = LoggedOut | LoggedIn 数据 PwdCheck = 正确 | 不正确
```

```
数据 ShellCmd : (ty : Type) -> Access -> (ty -> Access) -> Type where
    密码:字符串 -> ShellCmd PwdCheck ?password_in ?password_out 注销:ShellCmd () ?logout_in ?logout_out
    GetSecret :ShellCmd 字符串 ?getsecret_in ?getsecret_out
```

```
PutStr : String -> ShellCmd () 状态 (常量状态)
纯 : (res : ty) -> ShellCmd ty (state_fn res) state_fn
(>>=) : ShellCmd a state1 state2_fn ->
    ((res : a) -> ShellCmd b (state2_fn res) state3_fn) ->
        ShellCmd b state1 state3_fn
```

填充以下类型的孔:

Password 读取密码并将状态更改为LoggedIn或LoggedOut,
取决于密码是否正确 Logout - 将状态从 LoggedIn 更改
为LoggedOut GetSecret -只要状态为 LoggedIn,就读取秘密消息

以下函数应检查您是否有正确答案:

```
session : ShellCmd () LoggedOut (const LoggedOut) session = do Correct <-
    Password wurzel
```

```
|不正确 => PutStr 密码错误
味精<-GetSecret
PutStr( 密码:           ++ 显示味精 ++ \n )
登录
```

以下函数不应进行类型检查:

```
sessionBad : ShellCmd () LoggedOut (const LoggedOut)
sessionBad = 密码 "root"
味精<-GetSecret
PutStr( 密码:           ++ 显示味精 ++ \n )
登录

noLogout : ShellCmd () LoggedOut (const LoggedOut)
noLogout = 正确 <- 密码 "wurzel"
|不正确 => PutStr 密码错误
味精<-GetSecret
PutStr( 密码:           ++ 显示味精 ++ \n )
```

2 将硬币插入第 13 章定义的自动售货机时,

机器可以拒绝硬币。您可以通过更改类型来表示这一点

MachineCmd 和 InsertCoin。 MachineCmd 中的操作可以更改状态
根据其结果:

数据 MachineCmd : (ty : Type) -> PlaceState -> (ty -> PlaceState) -> Type

然后, InsertCoin 可以返回投币是否成功,

相应地更改状态:

```
InsertCoin :MachineCmd CoinResult (磅,冲击)
(\res => 案例 res 的
插入 => (S 磅,巧克力)
拒绝=> (磅,巧克力))
```

定义 CoinResult 类型,然后在 Vending 中对 MachineCmd 进行此更改

.idr。此外,细化其他命令的类型和执行

machineLoop 根据需要。

14.3 一个经过验证的猜谜游戏:用类型描述规则

作为本章的一个总结性示例,我们将看看如何使用类型来精确地表示游戏规则,并确保游戏的任何实现

正确遵守规则。我们将重温第 9 章中的一个例子,猜词

游戏刽子手。

回顾一下它是如何工作的,您定义了一个 WordState 类型来表示游戏。 WordState 定义如下,包括猜测的数量和剩余的字母作为参数:

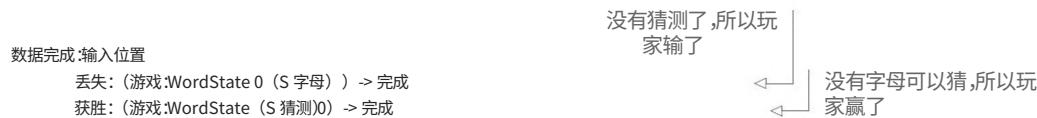
```
数据 WordState : (guesses : Nat) -> (letters : Nat) -> Type where
  MkWordState: (字:字符串)
    -> (缺少:Vect 字母字符)
    -> WordState guesses_remaining letters
```

目标词,由玩家猜

字母仍有待玩
家猜

一个经过验证的猜谜游戏：以类型描述规则

您还定义了一个`Finished`类型来表示游戏何时完成，或者是因为单词中没有字母可以猜（所以玩家赢了），或者没有剩余的猜测（所以玩家输了）：



鉴于这些，您定义了一个名为`game` 的主循环，它接受一个包含猜测和字母的`WordState`，并循环直到游戏完成：

```
game : WordState (S guesses) (S letters) -> IO Finished
```

在实现中，您使用类型来帮助引导您找到有效的实现。但是您也可能使用这种类型编写了错误的游戏实现。例如，以下游戏的实现也可以很好地键入，但错误，因为它在所有情况下都返回失败的游戏状态：

```
game : WordState (S guesses) (S letters) -> IO Finished game state = pure (Lost (MkWordState ANYTHING [ A ]))
```

虽然类型可以让您精确地表达游戏状态并帮助您为中间操作（例如处理猜测）赋予类型，但它并不能保证实现正确地遵循游戏规则。在前面的实现中，玩家是不可能获胜的！

在本节中，我们将在状态类型中精确地定义游戏规则，而不是定义`WordState`类型然后相信该游戏将正确地遵循游戏规则。正如`DoorCmd`表示何时可以对门执行操作，而`ATMCmd`表示何时可以对ATM 执行操作一样，我们可以定义一个依赖的`GameCmd`类型，该类型表示何时可以在游戏中执行特定操作，以及效果如何对那些操作会。与门和 ATM 示例一样，我们将首先定义状态以及可以在这些状态上执行的操作。

14.3.1 定义一个抽象的游戏状态和操作

首先，我们将考虑如何抽象地定义游戏规则，而不用担心实现的细节。游戏可以处于以下状态之一：

`NotRunning` 当前没有正在进行的游戏。要么游戏还没有开始，还没有字可猜，要么游戏已经完成。`Running` 游戏正在进行中，玩家还有许多猜测和字母要猜测。

在`Running` 的情况下，我们将使用剩余的猜测数和字母数来注释状态，就像我们之前对`WordState` 所做的那样，因为这意味着我们将能够

准确描述游戏何时赢(没有要猜的字母)或输(没有剩下的猜测)。我们可以用以下数据类型表达可能的状态:

```
数据 GameState : 输入位置
    NotRunning : 游戏状态
    运行: (猜测:Nat) -> (字母:Nat) -> GameState
```

然后,我们将支持一些用于操纵游戏状态的基本操作:

NewGame - 使用单词初始化游戏供玩家猜测	Guess - 允许玩家猜测字母
字母 Won - 宣布玩家赢得游戏	Lost - 宣布玩家输掉游戏

图 14.5 说明了这些基本操作如何影响游戏状态。输赢还有额外的前提条件:如果没有剩下的字母可以猜,我们只能宣布玩家赢了,如果没有剩下的猜测,我们只能宣布玩家输了。

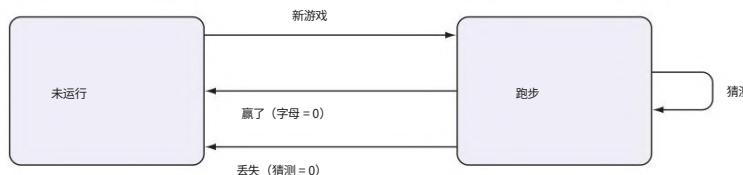


图 14.5 Hangman 的状态转换图。Running 状态还保存剩余的字母数和猜测数。Won 要求字母数为零, Lost 要求猜测数为零。

下一步是在依赖类型中精确地表示这些状态转换,包括关于每个操作有效所需的猜测数和字母数的特定规则。

14.3.2 定义游戏状态类型

我们将定义一个 GameCmd 类型,描述您可以运行的影响 GameState 的可能操作。

下一个清单显示了图 14.5 中的 NewGame、Won 和 Lost 的类型。像往常一样,包含 Pure 和 (\geq) 以便您可以引入纯值和序列操作。

清单 14.7 开始定义 GameCmd (Hangman.idr)

```
导入数据.Vect
如果任何定义不完整,则报告错误
    %默认总计
    data GameCmd : (ty : Type) -> GameState -> (ty -> GameState) -> Type where
        稍后您将使用 Vect
        代表缺失的字母。
```

一个经过验证的猜谜游戏:以类型描述规则



您可以使用字母获取单词中的不同字母,将单词转换为大写,然后将其转换为List Char,最后删除所有重复元素:

字母 : String -> List Char
letters str = nub (map
toUpper (unpack str))

nub 在 Prelude 中定义;它从列表中删除重复的元素。

清单 14.8 向 GameCmd 添加了一个 Guess 操作。Guess 的类型有一个前置条件和一个后置条件来解释猜测如何影响游戏: 作为前置条件,必须至少有一个可用的猜测 (S 个猜测) 和至少一个仍然可以猜测的字母 (S 个字母),或者试图猜测一个字母不会进行类型检查。

作为后置条件,如果猜测不正确,猜测的次数会减少,
如果猜对了,字母的数量就会减少。

清单 14.8 添加一个猜测操作 (Hangman.idr)

数据 GuessResult = 正确 | 不正确
 |—— 猜测的结果是正确的或不正确的。

```

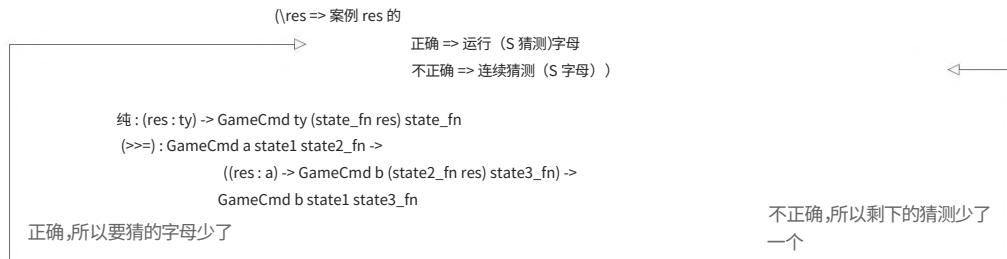
data GameCmd : (ty : Type) -> GameState -> (ty -> GameState) -> Type where
    新游戏: (字:字符串) ->
        GameCmd () NotRunning (const  
(Running 6 (length (字母词))))
    获胜:GameCmd () (Running (S guesses) 0) (const NotRunning)

    Lost : GameCmd () (Running 0 (S猜测)) (const NotRunning)

```

猜猜: (c:Char) ->
GameCmd GuessResult
(跑步 (S猜测) (S字母))

如果还有猜测和字母,您只能猜测。



最后,为了能够使用用户界面实现游戏,您需要添加用于显示当前游戏状态、显示任何消息以及读取用户猜测的命令:

```

data GameCmd : (ty : Type) -> GameState -> (ty -> GameState) -> Type where
  { 续清单 14.8 }
  ShowState : GameCmd() 状态 (常量状态)
  消息:字符串 -> GameCmd () 状态 (常量状态)
  ReadGuess : GameCmd 字符状态 (常量状态)
  
```

显示游戏状态应显示目标单词中的已知字母和剩余的猜测次数。例如,如果目标词是TESTING并且您已经猜到了T,还剩下六次猜测, ShowState应该显示以下内容:

T---T---

剩下6个猜测

当您实际实现游戏时,支持无限长的游戏循环很有用。例如,一旦您完成游戏,玩家可能想要开始新游戏。

下一个清单定义了一个GameLoop类型,使用Inf表示执行可能会无限期地继续。

清单 14.9 一种用于描述潜在无限游戏循环的类型 (Hangman.idr)

命名空间循环
 数据 GameLoop : (ty : Type) -> GameState -> (ty -> GameState) -> Type where
 (>>=) : GameCmd a state1 state2_fn -> ((res : a) -> Inf (GameLoop b
 (state2_fn res) state3_fn)) ->
 GameLoop b state1 state3_fn
 退出:GameLoop() NotRunning (const NotRunning)

引入一个新的命名空间,因为您正在
重载 (>>=)

您无法退出仍在运行的游
戏。

您可以使用GameLoop和GameCmd中的操作来定义以下函数,该函数实现了一个游戏循环:

```
gameLoop : GameLoop () (Running (S guesses) (S letters)) (const NotRunning)
```

一旦你有一个类型良好的、完整的gameLoop 实现,你就会知道它是规则的有效实现。游戏和玩家都无法

通过违反GameCmd中定义的规则来作弊。您只能在正确初始化的游戏上调用gameLoop，猜一个词，并且任何实现都必须是游戏的完整实现，因为完成GameLoop的唯一方法是调用Exit，这需要游戏处于NotRunning状态。

14.3.3 实现游戏

我们将以交互方式实现gameLoop并通过检查类型来查看游戏状态如何进行。

首先，您可以创建一个骨架定义，将类型中的猜测和字母带入范围，因为稍后您需要检查它们以检查玩家的进度：

```
gameLoop : GameLoop () (Running (S guesses) (S letters)) (const NotRunning) gameLoop {guesses} {letters} = ?gameLoop_rhs
```

要实现gameLoop，请执行以下步骤：

1细化 在gameLoop的每次迭代开始时，您将使用ShowState显示游戏的当前状态，读取用户的猜测，然后检查它是否正确：

```
gameLoop : GameLoop () (Running (S guesses) (S letters)) (const NotRunning) gameLoop {guesses} {letters} = do
```

```
    显示状态
    g <- ReadGuess ok <-
    Guess g ?
    gameLoop_rhs
```

2类型定义 如果您现在检查?gameLoop_rhs的类型，您会看到游戏的当前状态取决于猜测是否正确：

```
字母 :Nat
猜测 : Nat g : Char ok :
GuessResult
```

```
gameLoop_rhs : 游戏循环()
    (情况正常
        正确 => 运行 (S 猜测)字母
        不正确 => 连续猜测 (S 字母) )
    (\value => 未运行)
```

为了取得进展，您需要检查ok以确定游戏处于哪个状态：

```
gameLoop : GameLoop () (Running (S guesses) (S letters)) (const NotRunning) gameLoop {guesses} {letters} = do
```

```
    显示状态
    g <- ReadGuess ok <-
    Guess g case ok of
```

```
        正确 => ?gameLoop_rhs_1 不正确 => ?
        gameLoop_rhs_2
```

3 Type, define 在`?gameLoop_rhs_1`中，猜对了，所以let的个数
剩余的ters减少了，您可以通过检查其类型看到：

```
好的猜测结果
字母:Nat
猜测:Nat g:Char
-----
gameLoop_rhs_1: 游戏循环()
(运行(S猜测)字母)(\value => NotRunning)
```

只有当字母和猜测都剩余时，您才能继续进行游戏循环，因为它的输入状态是正在运行(S猜测)(S字母)。要决定如何继续，您需要检查字母的当前值：

```
情况正常
正确 => 大小写字母
Z => ?gameLoop_rhs_3 S k => ?
gameLoop_rhs_4 不正确 => ?
gameLoop_rhs_2
```

4 精炼 在`?gameLoop_rhs_3`中，没有字母可以猜，所以玩家赢了。您可以用`Won`声明玩家已获胜，进入`Not Running`状态。然后显示最终状态并退出：

```
情况正常
正确 => 大小写字母
Z => 赢了
显示状态
出口
S k => ?gameLoop_rhs_4 不正确 => ?
gameLoop_rhs_2
```

您需要明确退出，因为退出是打破GameLoop 的唯一方法。您只能退出处于`NotRunning`状态的游戏。

5 优化 在`?gameLoop_rhs_4`中，仍有字母要猜，因此您可以显示一条消息并继续`gameLoop`：

```
情况正常
正确 => 大小写字母
Z => 赢了
显示状态
出口
S k => 做消息“正确”
gameLoop 不
正确 => ?gameLoop_rhs_2
```

`Incorrect case`的工作原理类似，检查猜测是否仍然可用，如果没有，则宣布玩家输了。下面的清单给出了完整的定义，供参考。

一个经过验证的猜谜游戏：以类型描述规则

清单 14.10 gameLoop (Hangman.idr) 的完整实现

```
gameLoop : GameLoop () (Running ($ guesses) ($ letters)) (const NotRunning) gameLoop {guesses} {letters} = do
    显示状态
    g <- ReadGuess ok <- Guess
    g case ok of
        正确 => 大小写字母
            Z => 赢了
            显示状态
            出口
            S k => 执行消息 “正确” 游戏循环
        不正确 => 案例猜测
            Z => 丢失
            显示状态
            出口
            (S k) => 做消息 “不正确”
            游戏循环
```

您还需要初始化游戏。例如，您可以编写一个函数来设置一个新游戏，然后启动 gameLoop：

```
hangman : GameLoop() NotRunning(const NotRunning) hangman = 做NewGame “测试” gameLoop
```

到目前为止，您只定义了一个描述游戏中动作的数据类型。 gameLoop 函数描述了遵循规则的有效 Hangman 游戏中的动作序列。为了运行游戏，您需要定义游戏状态的具体表示和将 GameLoop 转换为一系列 IO 操作的函数。

14.3.4 定义一个具体的游戏状态

在第 13 章的堆栈示例中，我们有一个抽象的堆栈状态（堆栈上的项目数），以及一个由适当长度的 Vect 表示的具体状态。同样， GameState 是游戏的抽象状态，仅描述游戏是否正在运行，如果是，则剩余多少猜测和字母。

为了运行游戏，您需要定义一个相应的具体游戏状态，其中包括特定的目标词以及哪些特定的字母仍有待猜测。

下面的清单定义了一个带有 GameState 参数的 Game 类型，表示与抽象游戏状态相关的具体数据。

清单 14.11 表示具体的游戏状态 (Hangman.idr)

```
数据游戏 : GameState -> 输入位置
    GameStart : 游戏未运行
    GameWon : (word : String) -> Game NotRunning
```



为Game定义一个Show实现很方便,以便您可以轻松显示游戏进度的字符串表示形式。

清单 14.12 Game (Hangman.idr) 的Show实现

您可以使用Game来跟踪具体的游戏状态。当你执行一个GameLoop,你将把一个具体的游戏状态作为输入,并返回一个结果更新的游戏状态:



您使用Fuel,因为运行潜在的循环。特别是,当您阅读Guess从播放器,唯一有效的输入是单个字母字符,所以你需要继续要求输入,直到它有效为止。

如果燃料耗尽, GameResult需要说明执行失败。否则,它需要存储操作的结果和新的状态。至关重要的是,类型新状态可能取决于结果;例如,可用的猜测次数取决于Guess返回的是正确还是错误。一个

因此, GameResult是以下之一:

- 执行游戏产生的结果和输出状态对,有
- 根据结果计算的类型
- 如果燃料耗尽,则会出现错误

您可以按如下方式定义GameResult :

与 GameCmd 的定义一样,
Game 的参数是根据结果 res 计算的。

一个经过验证的猜谜游戏：以类型描述规则

`outstate_fn`包含在 `GameResult` 的类型中，因为您在输入您如何计算 `Game` 的输出状态。

现在您有了一个数据类型来表示游戏的具体状态将抽象状态作为参数，连同结果的表示，您已准备好实施运行。

14.3.5 运行游戏：执行 `GameLoop`

下一个清单概述了 `GameLoop` 的运行定义。这使用了另一个功能，`runCmd`，执行 `GameCmd`。目前 `runCmd` 的定义有一个漏洞。

清单 14.13 运行游戏循环 (`Hangman.idr`)

运行命令。我们将为此留下一个漏洞，并很快定义。

→ `runCmd : 燃料 ->`

游戏状态 -> `GameCmd ty instate outstate_fn ->`

`IO (GameResult ty outstate_fn)`

`runCmd 燃料状态 cmd = ?runCmd_rhs`

运行 :Fuel -> `Game instate -> GameLoop ty instate outstate_fn ->`

`IO (GameResult ty outstate_fn)`

跑干 = 纯 `OutFuel`

运行（更多燃料）`st (cmd >> next)`

= 执行 OK `cmdRes newSt <- runCmd Fuel st cmd`

| `OutFuel => 纯 OutFuel`

运行燃料 `newSt (下一个 cmdRes)`

运行（更多燃料）`st Exit = pure (OK () st)`

运行成功后返回运行结果

(此处为 `cmdRes`) 和更新状态 (此处为 `newSt`)。

第一个命令燃料耗尽

在运行中，当它成功时，您使用 `pure` 返回一对结果和新的状态。因为执行命令时经常会以这种形式返回结果，您可以定义一个辅助函数，好的，以使其更简洁：

好的：`(res:ty) -> 游戏 (outstate_fn res) ->`

`IO (GameResult ty outstate_fn)`

`ok res st = 纯 (OK res st)`

使用 `ok`，您可以将 `run` 的最后一个子句细化为以下内容：

运行（更多燃料）`st Exit = ok () st`

清单 14.14 给出了 `runCmd` 的概要定义，为 `Guess` 和阅读猜测案例。在其他情况下，您可以使用 `ok` 根据根据需要键入并执行 `IO` 操作。

清单 14.14 `runCmd` (`Hangman.idr`) 的大纲定义

`runCmd : Fuel -> Game instate -> GameCmd ty instate outstate_fn ->`

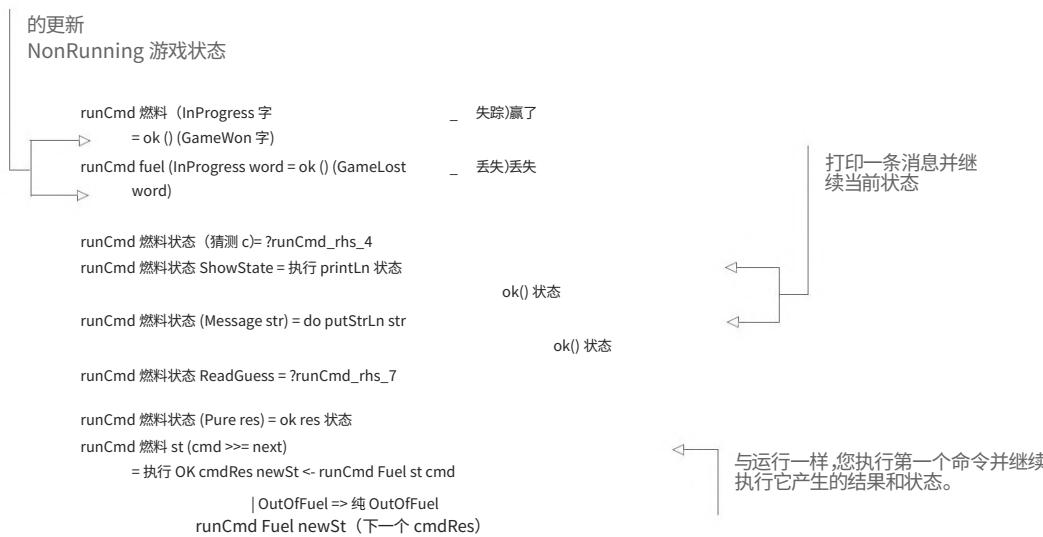
`IO (GameResult ty outstate_fn)`

`runCmd 燃料状态 (NewGame 字)`

= `ok () (InProgress (toUpper word)`

- `(fromList (字母单词))`

使用字母创建一个新的进行中的游戏
从单词中提取唯一的字母



当游戏正在进行时,游戏状态使用Game的InProgress构造函数,
它具有以下类型:

*剑子手> :t 进行中
进行中:字符串 ->
(猜测:Nat) ->
Vect 字母 Char -> 游戏 (运行猜字母)

第三个参数是仍有待猜测的字母向量。所以,在
猜测情况,您检查猜测的字符是否在缺失字母的向量中:

runCmd 燃料 (InProgress 字 = case isElem c 缺失) -> 失敗) (猜c)
--> Yes prf => ok Correct (InProgress word (removeElem c missing))
--> No contra => ok 不正确 (InProgress 字缺失)

正确,所以从
缺失字母的向量
返回 c 是否在缺失字母向量中的证
明

您在第 9 章的早期实现中以交互方式定义了removeElem
刽子手。为方便起见,我在这里重复一遍:

```

removeElem : (value : a) -> (xs : Vect (S n) a) ->
{auto prf : Elem 值 xs} ->
Vect na

removeElem value (value :: ys) {prf = Here} = ys
removeElem {n = Z} value (y :: []) {prf = There later} = 荒谬之后
removeElem {n = (S k)} value (y :: ys) {prf = There later}
= y :: removeElem 值 ys
  
```

一个经过验证的猜谜游戏:以类型描述规则

最后,您需要定义ReadGuess案例,它从播放器。输入仅在它是字母字符时才有效,因此您循环直到玩家输入一个有效的输入:

```
runCmd (更多燃料) st ReadGuess = do
    putStr 猜测:
    输入<getLine
    案例解包输入
    [x] => if isAlpha x
        然后 ok (toUpperCase x) st
        否则做 putStrLn 无效输入
        runCmd fuel st ReadGuess
    - => 做 putStrLn 无效输入
        runCmd fuel st ReadGuess
runCmd 干 ... = 纯燃料不足
```

如果用户继续输入无效输入,这种情况可能会无限循环,所以runCmd将Fuel作为参数,并在出现无效时消耗燃料输入。结果, runCmd本身保持完整,因为它要么消耗燃料,要么在每个递归调用上处理一个命令。runCmd是总的很重要,因为这意味着您知道执行GameCmd将继续取得进展,只要有要执行的命令。

您现在可以编写主程序,使用forever来确保,在实践中,跑步永远不会耗尽燃料。将以下内容添加到 Hangman.idr 的末尾:

```
%默认部分
永远:燃料
永远=永远更多

主要:我 ()
main = 永远运行 GameStart hangman
    纯的 ()
```

您现在应该可以在REPL 上执行游戏了。这是一个例子:

```
*剑子手> :exec
-----
剩下6个猜测
猜测:t
正确的
T---T---
剩下6个猜测
猜测:x
不正确
T---T---
剩下 5 个猜测
猜测:g
正确的
```

```
T--T--G  
剩下 5 个猜测  
猜测:不好  
输入无效  
猜测:
```

在此示例中,我们将规则的描述 (在GameCmd和GameLoop中)与规则的执行 (在runCmd和run 中)分开。本质上, GameCmd和GameLoop定义了一个接口,用于构建一个有效的 Hangman 游戏,正确地遵循规则。任何使用这些类型的类型良好的总函数都必须是规则的正确实现,否则就不会进行类型检查!

14.4 总结

您可以通过使用操作的结果从环境中获得反馈
计算命令的输出状态。
系统在运行命令后可能会处于不同的状态,这取决于命令是否成功。

定义操作的先决条件允许您以类型表示安全属性,例如当ATM 取款有效时。 系统可能会根据用户的输入在当前环境中是否有效而改变状态,例如PIN或密码是否正确。 谓词和自动隐式帮助您描述操作的有效输入状态

恰恰。
你可以用一个类型精确地描述一个游戏的规则,这样一个函数
类型检查必须是规则的有效实现。
您使用抽象状态类型来描述操作做什么,并根据抽象状态使用具体状态来描述它们相应的实现。

类型安全的并发编程

本章涵盖

使用并发原语定义用于描述并发进程的类型
使用类型来确保并发进程一致地通信

在 Idris 中， IO ()类型的值描述了与用户和操作系统交互的一系列动作，运行时系统按顺序执行这些动作。也就是说，它一次只执行一个动作。您可以将一系列交互操作称为一个过程。

除了在单个进程中按顺序执行操作外，能够同时、并发地执行多个进程并允许这些进程相互通信通常很有用。在本章中，我将介绍 Idris 中的并发编程。

消息传递并发编程是一个很大的话题，有几种方法可以写满他们自己的书。我们将看一些消息传递并发的小例子，其中进程

通过相互发送消息进行交互。Idris 运行时系统作为原语支持消息传递。实际上,向进程发送消息并接收到对应于调用以面向对象语言返回结果的方法。

并发编程有几个优点： 您可以在运行大型计算时继续

与用户交互。例如,用户可以在下载大文件时继续浏览网页。您可以显示在另一个进程中运行的大型计算的进度反馈,例如显示下载进度条。您可以充分利用现代CPU 的处理能力,将工作分配给运行在不同CPU内核上的多个进程。

本章介绍了类型驱动开发的一个更大的示例。首先,我将介绍 Idris 中并发编程的原语,并描述一般并发进程中可能出现的问题。然后,我将介绍一种用于描述并发进程的类型的初步尝试。这个最初的尝试会有一些缺点,因此我们将对其进行改进并得出一种允许进程安全且一致地相互通信的类型。

15.1 Idris 中并发编程的原语

Idris 基础库提供了一个模块 System.Concurrency.Channels,它包含用于启动并发进程和允许这些进程相互通信的原语。从理论上讲,这使您可以编写能够有效利用CPU并且即使在执行复杂计算时也能保持响应的应用程序。

但是,尽管有它的优点,并发编程是出了名的容易出错。多个进程相互交互的需要大大增加了程序的复杂性。例如,如果您在文件下载时显示进度条,则下载文件的进程需要与显示进度条的进程协调,以便知道下载了多少文件。这种复杂性导致程序在运行时失败的新方法:

死锁 两个或多个进程在它们可以继续之前互相等待执行某些操作。

竞态条件 系统的行为取决于行为的顺序
多个并发进程。

死锁的影响是相关进程冻结,不再接受输入或提供输出。如果客户端在等待从服务器接收消息的同时服务器正在等待从客户端接收消息,则两个并发进程 (我们称它们为客户端和服务器) 可能会死锁。如果发生这种情况,客户端和服务器都将冻结。

Idris 中并发编程的原语

竞争条件可能更难识别。图 15.1 中客户端的伪代码
 服务器说明了一个竞争条件,其中共享变量var的值
 取决于并发操作的执行顺序。我们假设
 Read和Write操作分别读取和写入共享可变对象的值
 变量,所以Read var读取共享变量var 的值。

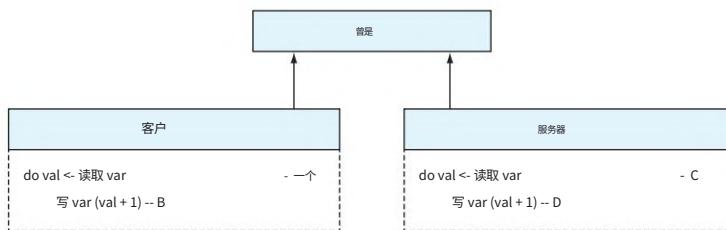


图 15.1 客户端和服务器的伪代码,其中每个进程读取和写入一个共享变量var。如果行 A 和 C 在行 C 和 D 之前执行,这可能会导致意外结果。

这里客户端和服务端并发执行, var的最终值取决于
 操作A、 B、 C 和D的执行顺序。 A将始终在之前运行
 B和C在D 之前,但除此之外,操作有六种可能的顺序。
 表 15.1 列出了这些排序和每种情况下var的结果值,假设
 初始值为1。

表 15.1 图 15.1 中每个操作序列的var值,给定var的初始值为1

操作顺序	var的值
A B C D	3
甲、丙、乙、丁	2
甲、丙、丁、乙	2
C、A、B、D	2
C、A、D、B	2
C、D、A、B	3

如表所示,当var的初始值为1 时,有两种可能的结果
 var,取决于读取和写入操作的运行顺序。这里,那里
 只是两个过程,每个过程有两个操作。随着程序的增长,
 这种不确定的结果变得更大。

在本章后面,使用我们前面讨论过的各种技术
 这本书,你将看到如何在 Idris 中编写并发程序,避免出现问题
 例如死锁和竞争条件。但首先,您需要了解原语

Idris 基础库为并发编程提供了
在编写需要相互协调的流程时会遇到的问题
其他。

15.1.1 定义并发进程

在一个完整的 Idris 程序中,类型为`IO()`的`main`函数描述了
运行时系统将在程序运行时执行的操作。因此,主函数描述了在单个进程中执行的动作。

我们使用的操作描述了控制台和文件I/O操作,但是
运行时系统还支持启动新进程和发送消息的操作
进程之间。以下操作有原始操作:

- 创建新流程。每个进程都与一个唯一的进程标识符相关联
(PID)。
- 向由其PID 标识的进程发送消息。
- 从另一个进程接收消息。

图 15.2 展示了一种我们可以使用消息传递来编写相互通信的并发进程的方法。在这个图中, `main` 和`adder`是两个 Idris
同时运行的进程,每个进程都可以向对方发送消息。
`main`发送给`adder`的消息使用以下类型:

数据消息 = 添加 Nat Nat

在这个例子中,在`main`向`adder`进程发送消息`Add 2 3`后,它期望
收到回复2和3 相加的结果,这样就可以并发使用
运行进程以实现响应其他进程发送的请求的服务。在这里,我们有一个执行加法的服务,在一个单独的
进程中运行。

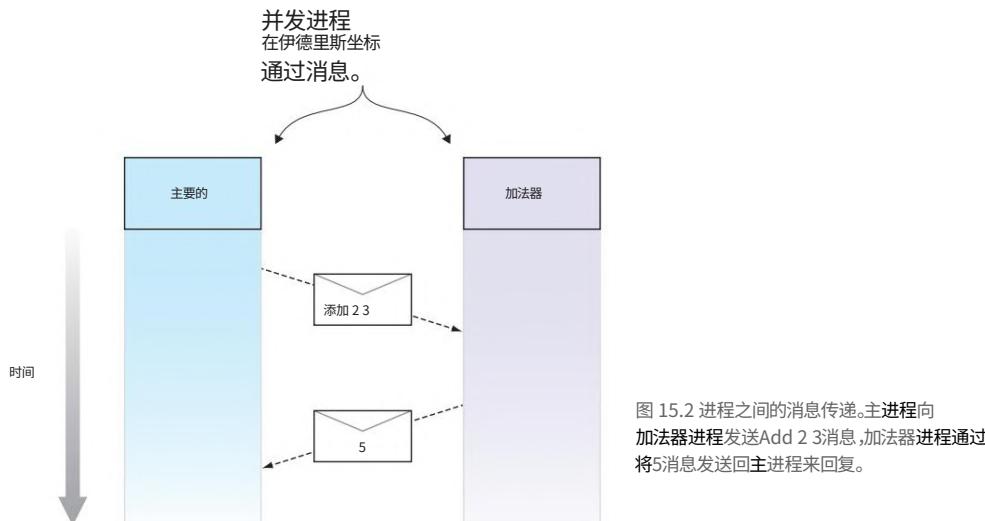


图 15.2 进程之间的消息传递。主进程向
加法器进程发送`Add 2 3`消息,加法器进程通过
将5消息发送回主进程来回复。

我将使用它作为并发进程的运行示例。接下来,您将看到如何使用 Idris 提供的并发原语在单独的进程中实现加法器服务,以及如何在 main.js 中使用该服务。

15.1.2 Channels 库:原始消息传递

基础库中的System.Concurrency.Channels模块定义了允许 Idris 进程创建新进程并相互通信的数据类型和操作。它定义了以下类型：

PID 表示进程标识符。每个进程都与一个 PID 相关联,该 PID 允许您设置与该进程的通信通道。

通道 两个进程之间的链接。它定义了一个通信通道,您可以沿该通道发送消息。

System.Concurrency.Channels模块还定义了以下用于创建新进程和设置通信通道的函数：

spawn 创建一个新进程来执行一系列IO () 类型的动作,
如果成功则返回其PID。

connect 启动与正在运行的进程的通信通道。 监听 等待另一个进程发起通信。
当另一个进程连接时,它会与该进程建立一个通信通道。

最后,该模块定义了在Channel上发送和接收消息的操作。我们将很快介绍System.Concurrency.Channels 中的类型定义,但首先让我们看看如何为main和adder进程设置进程和通道。主要过程如下： 1使用spawn 启动adder进程2使用connect设置与 adder 的通信通道3在通道上发送Add 2 3消息4在通道上接收带有结果的回复

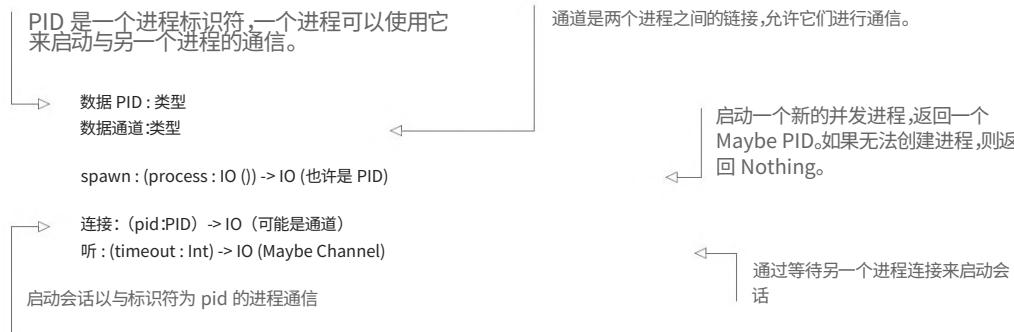
相应地,加法器过程如下工作：

- 1等待另一个进程使用listen发起通信并设置
另一个进程连接时的新通信通道
- 2在包含要添加的号码的频道上接收消息
- 3在频道上发送一条消息并附上结果
- 4返回步骤1,等待下一个请求

adder进程提供了一个长时间运行的服务,它等待传入的请求并向这些请求发送回复。一旦main启动了 adder进程,任何其他进程也可以向adder发送请求,只要它知道adder 的PID。

第一个清单显示了System.Concurrency.Channels中通道和PID的类型声明以及可用于创建进程和设置通信通道的函数。

清单 15.1 通道和 PID（在System.Concurrency.Channels 中定义）



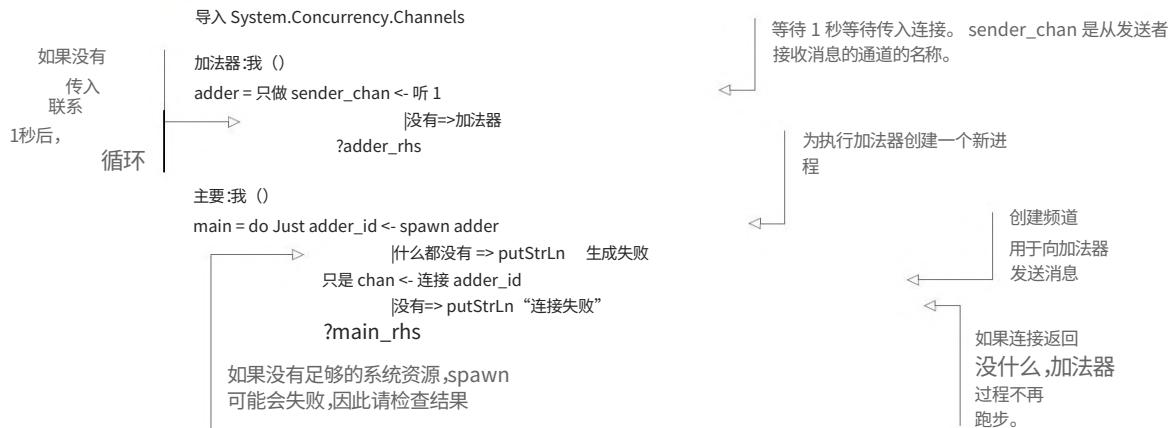
当您侦听连接或连接到另一个进程时,无法保证您会成功设置通信通道。可能没有

传入连接,或者您正在连接的进程可能不再运行。

所以,为了捕捉失败的可能性,连接和监听返回一个类型的值也许是频道。

您可以使用spawn、listen和connect将adder和main设置为单独的进程。下面的清单概述了一个设置进程的程序,离开相互发送消息的进程部分的漏洞。

清单 15.2 设置加法器进程的程序概要 (AdderChannel.idr)



现在您已经设置了进程和通道, main可以向adder发送消息,加法器可以回复。您可以使用以下原语:

unsafeSend 发送任何类型的消息

unsafeRecv 接收预期类型的消息

顾名思义,这些原语是不安全的,因为它们不提供一种方法

检查发送者和接收者是否期望消息以特定的方式发送

顺序,或者他们正在发送和接收一致类型的消息。尽管如此,目前我们将使用它们来完成main和adder 的实现。稍后,在 15.2 节中,您将看到如何制作安全版本,以确保进程使用一致的协议发送和接收消息。

为什么要支持不安全的通道类型? Idris 是一种旨在支持类型驱动开发的语言,它支持这种不安全的并发原语而不是更复杂的东西,这似乎令人惊讶。原因在于以类型驱动的方式实现安全并发程序有很多可能的方法,并且通过提供不安全的底层原语,Idris 不仅限于其中一种。您很快就会看到一种这样的方法。

以下清单显示了在System.Concurrency.Channels 中定义的这些原始操作的类型声明。

清单 15.3 原始消息传递 (在System.Concurrency.Channels 中定义)

```
unsafeSend : Channel -> (val : a) -> IO Bool unsafeRecv : (expected : Type) -> Channel
-> IO (也许是预期的)
```

在下一个清单中,您可以通过接收发送方的请求然后发送回复来完成adder的定义。使用unsafeRecv,您断言请求是Message 类型。

清单 15.4 加法器的完整定义 (AdderChannel.idr)

```
数据消息 = 添加 Nat Nat
adder : IO () adder = do
  Just sender_chan <- listen 1
  | 没有=>加法器
    只是 msg <- unsafeRecv 消息 sender_chan
    |什么都没有 => 加法器情况味精
      在通道上发送回复,
      其中包含 Nat 类型的
      输入总和
      添加 xy => ok <- unsafeSend sender_chan (x + y)
      加法器
```

Message 是 adder 期望接收
的类型。

等待您刚刚创建的频道上的
消息,类型为 Message

类似地,下面的清单显示了main 的完整定义,它使用unsafeSend发送消息并接收带有unsafeRecv 的Nat 类型的回复。

清单 15.5 main (AdderChannel.idr)的完整定义

```
main : IO () main = do
  Just adder_id <- spawn adder
  | 什么都没有 => putStrLn "生成失败"
    只是 chan <- 连接 adder_id
    | Nothing => putStrLn "Connection failed" ok <- unsafeSend chan (Add 2 3)
      等待频道上的回复,
      类型为 Nat
      | 需要回答 <- unsafeRecv Nat chan
        在您刚刚创建的频道上发送
        消息,类型为 Message
```

```
|什么都没有 => putStrLn  发送失败  putStrLn 答案
    <----- 打印从加法器收到的结果
```

如果你在REPL 中使用:exec编译和执行main ,你会看到它从adder接收到结果5 :

```
*AdderChannel> :exec main
5
```

这只是因为您已确保main和adder就通信模式达成一致。当main在通道上发送消息时,加法器进程期望在其相应通道上接收消息,反之亦然。

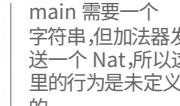
15.1.3 通道问题:类型错误和阻塞

通道提供了一种在进程之间建立链接并通过这些链接发送消息的原始方法。然而, unsafeSend和unsafeRecv的类型并没有提供任何关于进程如何相互协调的保证。

结果,很容易出错。

例如, adder发送一个Nat来回复main,但是如果main期望接收一个String 怎么办?

```
main : IO () main = do
Just adder_id <- spawn adder
    |什么都没有 => putStrLn 生成失败
    只是 chan <- 连接 adder_id
    | Nothing => putStrLn Connection failed  ok <- unsafeSend chan (Add 2 3)
回答 <- unsafeRecv String chan
    |什么都没有 => putStrLn  发送失败  putStrLn 答案
```



在这种情况下,执行main的行为将无法预测,并且很可能会崩溃,因为在运行时接收到的消息类型与预期类型之间存在不一致。 unsafeSend和unsafeRecv类型中没有任何内容解释发送和接收是如何在两个进程之间协调的,因此 Idris 很乐意接受main为有效,即使在这种情况下协调是不正确的。

如果unsafeSend和unsafeReceive操作在每个进程中不对称,则会出现不同的问题。例如, main可能会在同一频道上发送第二条消息并期待回复:

```
main : IO () main = do
Just adder_id <- spawn adder
    |什么都没有 => putStrLn 生成失败
    只是 chan <- 连接 adder_id
    | Nothing => putStrLn Connection failed  ok <- unsafeSend chan (Add 2 3)

只需回答 <- unsafeRecv Nat chan
    |什么都没有 => putStrLn  发送失败  putStrLn 答案
```



```
ok <- unsafeSend chan (Add 3 4)
```

```
只需回答 <- unsafeRecv Nat chan
|什么都没有 => putStrLn "发送失败" putStrLn 答案
```

此处执行阻塞,因为 adder 仅
回复通道上的第一条消息。

即使此类型检查成功,当您尝试执行它时,它也会打印来自adder的第一个回复,但在等待第二个回复时会阻塞。adder使用listen创建通道后,它只回复该通道上的一条消息。

尽管通道本身是不安全的,但您可以将它们用作定义类型安全通信的原语。我们将定义一个类型来描述通信进程之间的协调,然后编写一个使用不安全原语执行描述的运行函数。即使最终您需要使用原语,您也可以将所有细节封装在一个单一的、描述性的类型中,然后您可以将其用于通信系统的类型驱动开发。

15.2 定义安全消息传递的类型在 Idris 运行时系统中,并发进程

彼此独立运行。

没有共享内存,进程相互通信的唯一方法是相互发送消息。因为没有共享内存,所以不存在同时访问共享状态引起的竞态条件,但还有其他几个问题需要考虑:

如何确保main发送的消息类型与adder期望接收的消息类型相同?如果main向 adder 发送消息,但adder没有回复,会发生什么? main发送消息时, adder停止运行怎么办? 如何防止main和adder都在等待对方的消息?

在本节中,您将了解如何通过定义Process类型来解决这些问题,该类型允许您描述类型良好的通信进程。

类型和并发编程对并发编程中的类型的支
持在主流编程语言中通常非常有限,但有一些例外,例如 Go 中的类型化通道。

一个困难是,除了通道可以承载的消息类型之外,您还需要考虑消息传递的协议。换句话说,除了要发送什么(类型)之外,您还需要考虑何时发送它(协议)。

尽管如此,对并发编程类型的研究还是进行了重要的研究,最著名的是始于 Kohei Honda 的 1993 年论文“Dyadic Interaction 的类型”的会话类型研究。我们将在本节中实现的类型是具有最小协议的会话类型的实例,其中客户端发送一条消息,然后接收一条回复。如果您有兴趣进一步探索,Bernardo Toninho 和 Nobuko Yoshida 最近(2016 年)的一篇论文“在多方会话类型中验证数据”描述了在并发程序中使用类型的更复杂的方法。

但是,我们不会在第一次尝试时就正确地实现这个。正如在类型驱动开发中经常出现的情况一样,我们会发现我们需要改进类型以解决在我们第一次尝试后变得明显的问题。我们将首先定义一个特定于加法器服务的类型,然后对其进行细化以支持保证无限期响应请求的通用服务。

15.2.1 在一个类型中描述消息传递过程

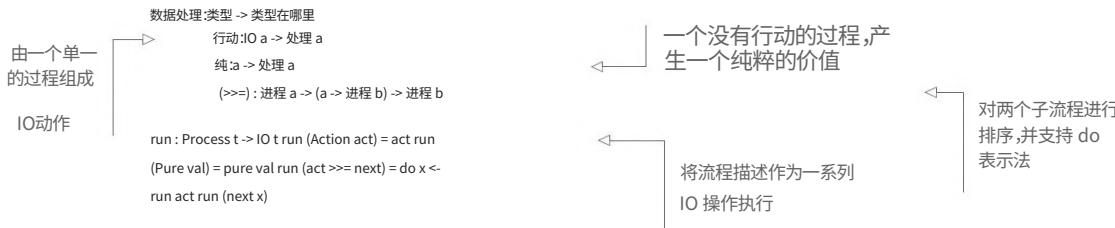
早些时候,我描述了原始Channel类型的两个问题,它们使这种原始形式的并发编程不安全:

无法检查对请求的响应是否具有正确的类型。在通信过程中无法检查发送和接收消息的对应关系。

我们将通过定义用于描述流程的类型,然后根据需要对其进行细化以支持我们需要的消息传递功能,从而解决这两个问题。

首先,您可以定义一个支持IO操作、构造纯值和排序的流程类型。

清单 15.6 描述流程的类型 (Process.idr)



在行动中使用IO通过在行动中使用IO,您可以在进程中包含任意IO操作,例如写入控制台或读取用户输入。这有点太笼统了,因为IO操作包括不安全的通信原语等。您可以通过定义更精确的命令类型来限制这一点(参见第11章的示例),但在此示例中我们将坚持使用IO。

目前, Process只不过是IO操作序列的包装器。

下一步是扩展它以支持生成新进程。您可以使用System.Concurrency.Channels中的PID定义数据类型来表示可以接收消息的进程:

数据 MessagePID = MkMessage PID

接下来,您可以向Process添加一个构造函数,该构造函数描述生成一个新进程并返回该进程的MessagePID(如果成功)的操作:

Spawn : Process () -> Process (也许是 MessagePID)

您还需要扩展run才能执行新的Spawn命令。这个使用spawn原语生成一个新进程,然后返回一个包含新PID的MessagePID :

```
run (Spawn proc) = do Just pid <- spawn (run proc)
                      |> 纯粹的没有
                      |> 纯 (只是 (MkMessage pid))
```

添加新构造函数请记住,在添加Spawn之后
构造函数,您可以通过将光标悬停在名称 run 上按 Ctrl Alt-A 来添加要在 Atom 中运行的缺失案例。

接下来,您可以添加命令以允许进程相互发送消息。在前面的例子,主进程发送Message类型的请求并等待Nat类型的相应回复。您可以将此行为封装在一个请求命令:

请求 :MessagePID -> 消息 -> 进程 (可能是 Nat)

返回Maybe Nat而不是Nat的原因是您没有任何保证MessagePID引用的进程仍在运行。当你运行一个

请求,您需要连接到为请求提供服务的进程,向其发送消息,然后等待回复:



在进程类型中封装原语你仍然需要使用 unsafeSend 和 unsafeRecv,但通过将它们封装在 Process 中
数据类型,你知道你的程序中只有一个地方使用
不安全的原语。您需要小心地正确定义此定义,但是一旦
你知道,你知道任何消息传递程序都是按照
Process 类型将正确地遵循消息传递协议。

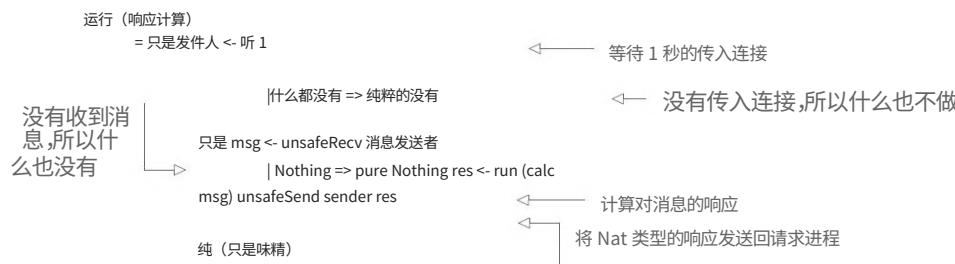
加法器进程等待传入消息,计算结果,并发送
回复请求者。您可以将此行为封装在单个响应中
命令:

响应 :((msg : Message) -> Process Nat) -> Process (Maybe Message)

这需要一个函数作为参数,当给定从一个接收到的消息时
请求者,计算发回的Nat。它返回一个类型为Maybe Message 的值,
如果它没有处理传入的消息,则它要么是Nothing,要么是Just

`msg`,如果它处理了传入的消息(`msg`)。如果您需要对传入消息进行任何进一步处理,即使在发送响应之后,这也很有用。

当您运行`Respond`命令时,您将等待一秒钟的消息,然后,如果有消息,则计算响应并将其发回:



响应的替代实现如果没有传入消息,响应案例的此实现将等待 1 秒。另一种更灵活的实现可能允许用户指定超时。例如,如果没有传入请求,则如果进程有其他工作要做,则继续等待可能没有意义。

清单 15.7 展示了如何使用`Process`定义`adder`和`main`。我们将它们称为 `procAdder`和`procMain`以将它们与早期版本区分开来。在`procAdder`中,您使用`Respond`来解释如何响应消息,而在`procMain`中,您使用`Request`将消息发送到生成的进程。

清单 15.7 实现一个类型安全的加法器进程 (Process.idr)

```

procAdder : Process () procAdder = do
  Respond (\msg => case msg of
    Continue --> procAdder
    WaitAnswer --> adder_id <- Spawn procAdder
    _              --> adder_id <- Request adder_id (Add 2 3)
    _              --> Nothing <- Action (putStrLn "请求失败")
    _              --> Action (printLn "答案")
  )
  adder_id <- spawn procAdder
  adder_id <- request adder_id (Add 2 3)
  nothing <- action (putStrLn "请求失败")
  action (printLn "答案")
  
```

通过发送响应 $x + y$ 来响应 Add xy 形式的消息

产生一个必须根据 Process 协议发送和接收消息的进程

发送请求。如果成功,则通过 answer 给出响应。

您可以在REPL 中尝试此操作,使用`run`将`procMain`转换为一系列I/O操作:

*进程> :exec run procMain 5

与以前的版本不同, `procMain`不能期望接收`String`而不是`Nat`,因为`Request`的类型不允许它。您还封装了

使用请求和响应的通道上的通信协议,因此您知道创建通道后不会发送或接收太多消息。

作为第一次尝试,这是对使用Channel的原始实现的改进,但是您可以通过多种方式对其进行改进。例如,procAdder不是总数:

```
*Process> :total procAdder Main.procAdder 由于递归
路径可能不是总的:Main.procAdder
```

这可能是一个问题,因为不完整的进程可能无法成功响应请求。作为第一个改进,您可以修改Process类型,并相应地修改run的定义,以便像proc Adder这样无限期运行的进程是总的。

15.2.2 使用 Inf 使进程总计

正如您在第 11 章中看到的,您可以使用Inf将部分数据标记为可能无限:

```
Inf : 类型 -> 类型
```

然后,如果一个函数在有限时间内产生类型良好的无限结果的构造函数的有限前缀,则可以说它是完全的。实际上,这意味着任何时候使用Inf类型的值,它都需要作为数据构造函数的参数或数据构造函数的嵌套序列。

您已经在第 11 章和第 12 章中看到了使用Inf定义潜在无限进程的各种方法。在这里,您可以通过将以下构造函数添加到 Process 来使用它来显式标记循环的进程部分:

```
循环:Inf (进程 a) -> 进程 a
```

作为参考,下一个清单显示了在新文件 ProcessLoop.idr 中定义的Process 的当前定义,包括Loop。

清单 15.8 新的Process类型,扩展了Loop (ProcessLoop.idr)

```
数据消息 = 添加 Nat Nat
```

←—— 进程可以发送的消息类型

```
数据 MessagePID = MkMessage PID
```

←—— 可以响应消息的进程的 PID

```
数据处理:类型 -> 类型在哪里
```

```
请求 :MessagePID -> 消息 -> 进程 (可能是 Nat)
响应 :((msg : Message) -> Process Nat) -> Process (Maybe Message)
Spawn : Process () -> Process (也许是 MessagePID)
循环:Inf (进程 a) -> 进程 a
行动:IO a -> 处理 a
纯:a -> 处理 a
(>>=) : 进程 a -> (a -> 进程 b) -> 进程 b
```

显式循环,执行潜在的无限进程

可以无限循环的进程的描述

使用Loop,您可以如下定义procAdder,明确指出对procAdder的递归调用是一个潜在的无限过程:

```
procAdder : Process () procAdder = do
    Respond (\msg => case msg of
        _ &gt;> return (x + y)
    )
    run procAdder
```

这个版本的procAdder总共是：

*ProcessLoop> :total procAdder Main.procAdder 是
Total

通过使用显式循环构造函数，您可以标记Process的无限部分，以便您至少可以确定任何无限递归都是有意的。此外，正如您将在下一节中看到的，它将允许您进一步细化Process，以便您可以准确控制何时允许流程循环。

您还需要扩展运行以支持循环。最简单的方法是直接执行动作

运行 (循环动作)= 运行动作

不幸的是,这个新的run定义并不是完全的,因为整体检查器(正确地!)不相信act是比Loop act更小的序列

*ProcessLoop> :total run Main.run 由于递归路径可能不是全部：
主运行,主运行,主运行

与第 11 章中的无限进程一样，您可以定义 Fuel 数据类型来给出明确的执行限制以运行。每次循环时，都会减少可用的燃料量。下面的清单显示了如何扩展 run 以便它在用完 Fuel 时终止，遵循您在第 11 章中已经看到的模式。

清单 15.9 新的运行函数,有执行限制 (ProcessLoop.idr)

数据燃料 = 干燥 | 更多 (懒惰的燃料)

```
run : Fuel -> Process t -> IO (Maybe t) = pure Nothing run Dry run  
chan <- connect phoebeRequestChan (Unsafe thing) msg unsafeSend  
chan msg if ok then do Just x <- unsafeRecv Nat chan
```

当进程耗尽燃料时返回
Nothing

| Nothing => pure (Just Nothing) pure (Just (Just x))
else pure (Just Nothing)

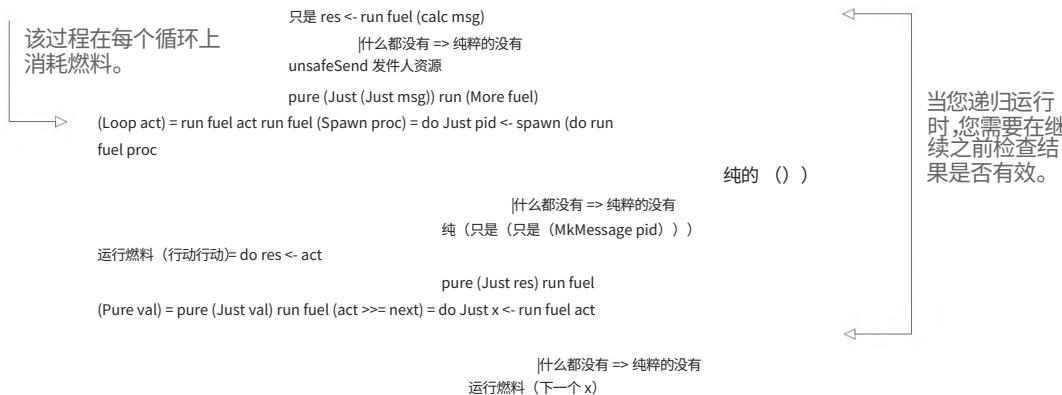
仅在过程
仍有燃料
时使用

运行燃料 (响应计算)= 只是发送者 <- 听 1

|什么都没有=>纯（只是什么都没有
只是 msg <- unsafeRecv 消息发送者
|什么都没有=>纯（只是什么都没有

定义安全消息传递的类型

417



请记住，您可以通过使用单个偏函数来生成无限量的Fuel并允许进程无限期地运行：

部分永远:燃料

永远=永远更多

使用单个永久函数来说明允许无限期进程运行多长时间意味着您可以最大限度地减少所需的非全部函数的数量。因为run是总的，所以您知道只要有要执行的操作，它就会继续执行流程操作。为方便起见，您还可以定义一个用于启动进程并丢弃其结果的函数：

部分 runProc :
 $\text{Process} () \rightarrow \text{IO} ()$ $\text{runProc proc} = \text{do run forever proc}$

纯的 ()

然后，您可以尝试如下执行procMain，它会像以前一样显示答案5：

*ProcessLoop> :exec runProc procMain 5

使用循环，您可以编写永远循环的进程，并且通过明确说明它们何时循环来完全循环。不幸的是，仍然不能保证循环进程会响应任何消息。例如，您可以按如下方式定义procAdder：

```
procAdderBad1 : Process () procAdderBad1 = do
Action (putStrLn "我今天不在办公室")
运行 procAdderBad1
```

甚至像这样：

```
procAdderBad2 : 处理 () procAdderBad2 = 循环
procAdderBad2
```

这两个程序都进行了类型检查，并且都进行了全面检查，但都不会响应任何消息，因为没有 Respond 命令。在 proc AdderBad2 的情况下，它是全部的，因为对 procAdderBad2 的递归调用是 Loop 构造函数的参数，因此它将产生构造函数的有限前缀。因此，完全使用 Loop 不足以保证进程会响应请求。

TOTALITY 的含义，在实践中 Totality 意味着您可以保证函数的行为与其类型所描述的完全一致，因此如果类型不够精确，那么保证也不是！对于 Process，类型不够精确，无法保证进程在任何循环之前包含响应命令。

此外，Process 类型特定于编写并发服务以添加数字的问题。如果你想编写不同的服务怎么办？您不想为您可能想要生成的每种服务编写不同的 Process 类型。

为了解决这些问题，您需要用另外两种方式来细化 Process 类型：在第 15.2.3 节中，您将细化它以确保服务器进程在循环的每次迭代中响应请求。在 15.2.4 节中，您将看到如何使用一等类型来允许 Process

回复任何类型的消息。

15.2.3 使用状态机和 Inf 保证响应

在第 13 章中，您看到了如何通过在类型中表示状态机来保证系统以正确的顺序执行必要的操作。像 adder 这样的服务器进程可能处于以下几种状态之一，具体取决于它是否接收并处理了请求：

NoRequest 它还没有为任何请求提供服务。已发送它已发送对请求的响应。完成 它已经完成了一个循环的迭代，并准备好服务于下一个请求。

图 15.3 说明了响应和循环命令如何影响进程的状态。

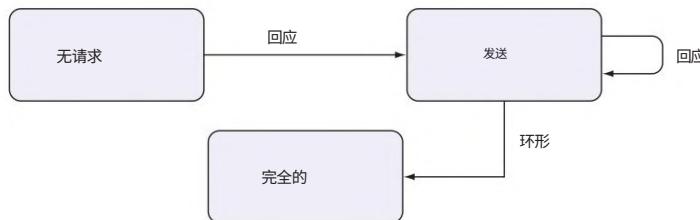


图 15.3 显示服务器进程中的状态和操作的状态转换图。一个进程以 NoRequest 状态开始，必须以 Complete 状态结束，这意味着它已经响应了至少一个请求。

如果您有一个以NoRequest状态开始并以Complete结束的进程状态,您可以确定它已经回复了请求,因为只有这样才能到达完成状态是通过调用响应。您还可以确定它正在继续接收请求,因为达到Complete状态的唯一方法是调用Loop。通过在其类型中表达进程的状态,您可以做出更有力的保证关于该过程的行为方式。

您可以细化Process的类型以表示执行过程之前和之后的状态:



类型中的状态给出了一个过程的前置条件和后置条件。为了例子:

Process () NoRequest Complete是响应某个进程的类型
请求然后循环。
Process () NoRequest Sent是响应一个或
更多请求,然后终止。
Process() NoRequest NoRequest是响应no的进程类型
请求然后终止。

清单 15.10 显示了精化的Process,其中每个命令的类型都解释了它如何影响整个过程状态。在这个定义中,在状态没有前提条件且状态没有变化的情况下,输入和输出状态都是st。

清单 15.10 用输入和输出状态注释 Process 类型 (ProcessState.idr)





返回请求类型之前,您发送了Message类型的请求并收到了Maybe Nat类型的响应。您使用 Maybe是因为您无法保证服务仍在运行,因此请求可能会失败。现在您已经设置了Process状态,以便服务始终响应请求。如果您发送一个请求,您可以保证在有限时间内得到Nat类型的响应。

当您使用这个新定义时,函数无法调用Loop ,除非该函数能够满足它已经发送了对请求的响应的先决条件。此外, Loop也是进程达到Complete状态的唯一途径。因此,您只能在保证循环的进程上调用Loop ,因为该进程必须以NoRequest状态开始并以Complete状态结束。

您仍然可以像以前一样定义procAdder ,因为每个命令都满足 pre 条件,它的类型现在声明它必须响应请求然后循环:

```
procAdder : Process () NoRequest Complete procAdder = do
  Respond (\msg => case msg of
    添加 xy => 纯 (x + y))
    运行 procAdder
```

然而,前面定义的两个错误版本不再进行类型检查,因为当您尝试使用这些命令时,它们不满足Process给出的先决条件。例如,您可以尝试以下定义:

```
procAdderBad1 : Process () NoRequest Complete procAdderBad1 = do Action
  (putStrLn 我今天不在办公室 )
  运行 procAdder
```

Idris 报错,因为Loop前没有Respond :

```
ProcessState.idr:63:21:
使用预期类型检查 procAdderBad1 的右侧时
  Process () NoRequest 完成

检查构造函数 Main.>>= 的应用程序时:类型不匹配
```

处理已发送完成 (循环类型_)
和
Process() NoRequest Complete (预期类型)

具体来说:
类型不匹配

发送
和
无请求

这个错误信息意味着当你调用Loop时,进程应该处于Sent状态,但此时它处于NoRequest状态,还没有发送任何响应。出于相同的原因,您将收到类似错误消息,其定义如下:

```
procAdderBad2 : Process () NoRequest Complete procAdderBad2 = Loop
procAdderBad2
```

为了使用精炼的Process执行程序,您需要修改run和runProc。首先,您需要修改它们的类型:

```
run:Fuel -> Process t in_state out_state -> IO (也许) runProc:Process () in_state out_state -> IO ()
```

定义大多与以前的版本相同。一个变化是在运行中请求案例的定义,现在您知道请求将始终在有限时间内收到回复:

```
run fuel (Request (MkMessage process) msg) = do Just chan <- connect
    process
        | _ => pure Nothing ok <-
        unsafeSend chan msg if ok then do Just x <-
            unsafeRecv Nat chan
                |什么都沒有 => 纯粹的沒有
                pure (Just x) else pure
                    Nothing
```

RUN 的返回值如果运行因任何原因失败,则返回Nothing。到目前为止,这只有在燃料耗尽的情况下才会发生。您现在已经设置了Process,以便协调发送者和接收者,因此,至少在理论上,通信不会失败。如果通信确实失败了,要么是run的实现出错,要么是更严重的运行时错误,所以在这种情况下你也可以返回Nothing。

您还需要修改procMain的类型,以与改进的Process类型保持一致。此类型明确声明procMain不打算响应任何传入请求,因为它以NoRequest状态结束:

```
procMain : Process () NoRequest NoRequest
```

为客户端(如procMain)和服务(如procAdder)定义类型同义词很方便。它们都使用Process,但它们在影响状态的方式上有所不同

过程:

```
服务:类型 -> 类型
服务 a = 处理 NoRequest 完成
```

```
客户端:类型 -> 类型
客户端 a = 处理 NoRequest NoRequest
```

下面的清单显示了procAdder和procMain的细化定义,使用这些类型的同义词用于客户端和服务器进程。

清单 15.11 procAdder和procMain (ProcessState.idn)的细化定义

<p>服务是一个以 NoRequest 状态开始并以 Complete 状态结束的 Process。</p> <pre>procAdder : Service () procAdder = do Respond (\msg => case msg of Add x y => pure (x + y))</pre> <p style="text-align: center;">运行 procAdder</p>	<p>客户端是一个以 NoRequest 状态开始和结束的进程。</p> <pre>procMain : Client () procMain = do Just adder_id <- Spawn procAdder Nothing => Action (putStrLn "Spawn failed") answer <- Request adder_id Add x y => Action (putStrLn ("Add " ++ show x ++ " " ++ show y)) _ => Action (putStrLn "Unknown request")</pre> <p style="text-align: center;">行动 (printLn 答案)</p>
---	--

如果你在REPL 上尝试这个,你会看到它像以前一样显示5 :

```
*ProcessState> :exec runProc procMain 5
```

您现在有了一个具有以下保证的Process定义,由 Process 定义中的前置条件和后置条件确保:

所有Message类型的请求都发送Nat 类型的响应。 每个以Spawn启动的进程都保证无限循环并响应
对每次迭代的请求。
因此,每次进程向使用Spawn启动的服务发送请求时,只要该服务由一个总函数定义,它就会在有限时间
内收到响应。

这意味着您可以编写不会死锁的类型安全的并发程序,因为每个请求都保证最终会收到响应。但在这个阶段,它只允许您编写一种服务 接收消息并返回Nat 的服务。如果您可以使用用户定义的发送者和接收者之间的交互来定义通用消息传递过程,那将会更加有用。正如您将看到的,您可以通过对Process 进行最后的小改进来实现这一点。

15.2.4 通用消息传递过程

当进程接收到Add x y 形式的请求时,它会发回Nat 类型的响应。您可以在类型级函数中表达请求和响应类型之间的这种关系:

```
AdderType : 消息 -> 类型
AdderType (加 xy) = Nat
```

这个函数描述了Process支持的接口:如果它收到一个Add x y 形式的消息,它将发送一个Nat 类型的响应。
您可以定义其他接口

这边走;例如,下面的清单描述了一个进程的接口,该进程响应对列表执行操作的请求。

清单 15.12 描述List操作的接口

```
数据 ListAction : 假设 where
    长度 : 列表元素 -> ListAction
    追加 : 列表元素 -> 列表元素 -> 列表动作

ListType : ListAction -> 类型
ListType (长度 xs) = Nat
ListType (Append {elem} xs ys) = List elem
```

由同时操作列表的进程接收的消息类型

← 获取列表的长度会产生一个 Nat。

← 附加两个元素类型为 elem 的列表会生成一个 List elem。

一般来说,进程的接口是一个类似于AdderType或ListType的函数,它从请求中计算出响应类型。除了定义进程可以发送和接收的特定类型之外,您还可以通过以下方式将接口作为进程类型的一部分包含在内

为接口添加一个附加参数,如图 15.4 所示。

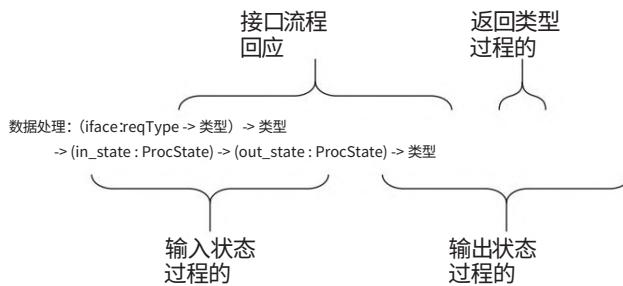


图 15.4 细化的流程类型,包括流程响应的接口作为类型的一部分

请求进程类型Process的iface参数包括一个类型变量, reqType。这是一个隐含的参数,它定义了进程可以接收的消息类型。Idris将从iface参数推断reqType。比如在procAdder中, iface就是AdderType,所以reqType必须

成为消息。

我们将很快讨论Process的精确定义。一旦定义, proc Adder服务将响应AdderType 定义的接口:

```
procAdder : Process AdderType() NoRequest Complete
```

某些进程,如procMain,不响应任何请求。你可以做这个通过定义它们的接口来显式地定义类型,如下所示:

```
NoRecv : 无效 -> 类型
NoRecv = 常量无效
```

请记住第8章中的Void是空类型,没有值。因为您永远无法构造Void类型的值,所以提供NoRecv接口的进程永远无法接收请求。您可以将它用于procMain的以下类型:

procMain : 处理 NoRecv () NoRequest NoRequest

您还需要重新定义类型同义词Service和Client以包含接口描述。一个Service有一个接口，但是一个Client没有收到请求：

服务：(iface:reqType -> 类型) -> 类型 -> 类型
Service iface a = 处理 iface a NoRequest Complete

客户端 : 类型 -> 类型
客户端 a = 处理 NoRecy a NoRequest NoRequest

当你创建一个新进程时，你会得到一个新进程的PID作为MessagePID。您应该只在消息与该进程的接口匹配时向该进程发送消息，因此您可以细化MessagePID以在其类型中包含它支持的接口：

数据 MessagePID : (iface : reqType -> Type) -> Type where
MkMessage : PID -> MessagePID iface

现在,如果你有一个MessagePID AdderType类型的PID ,你知道你可以向它发送 Message 类型的消息,因为那是AdderType 的输入类型。

将所有这些放在一起，您可以改进Process以描述它自己的接口，并明确何时将特定类型的请求发送到另一个Process是安全的。下一个清单显示了Request、Respond和Spawn的细化类型。

清单 15.13 细化 Process 以将其接口包含在类型中, 第 1 部分 (ProcessIFace.idr)

数据 ProcState = NoRequest |已发送 |完全的

数据处理:(iface : reqType -> Type) -> Type -> (in_state : ProcState) -> req
有类型(out_state : ProcState) -> service_reqType,
并且type where service_iface 有类型Request:
service_reqType -> 类型。 (msg : service_reqMessageID<service_iface>
(msg : reqType) -> Process iface(iface msg) NoRequest NoRequest msg) ptl.Respond(Maybe
reqType) st Sent

具有由
service_iface 描
述的接口的进程的
PID

Spawn : Process service_iface () NoRequest Complete ->
Process iface (Maybe (MessagePID service_iface)) st st {-- 继续清单 15 14 --}

来自服务的回复将通过对请求应用 service_iface 来计算，以确保回复类型与请求相对应。

如果您使用 service_iface 描述的接口生成服务器，则 PID 的类型为 MessagePID
service_iface

如果收到req消息，则需要的响应类型由iface req计算。

定义安全消息传递的类型

下面的清单完成了Process 的细化定义,添加了Loop、 Action、 Pure和(>=)。在每种情况下,您需要做的就是添加一个iface参数

过程。

清单 15.14 细化Process以将其接口包含在类型中,第 2 部分 (ProcessIFace.idr)

数据处理: (iface : reqType -> Type) -> Type -> (in_state : ProcState) -> (out_state :
ProcState) -> Type where {-- 继续清单 15.13 --}

在 Loop 和 (>=) 中,类型明
确声明接口不会改变。

```
Loop : Inf (Process iface a NoRequest Complete) ->  
        处理已发送完成的 iface  
操作:IO a -> 处理 iface a st st  
Pure : a -> Process iface a st st  
(>=) : 处理 iface a st1 st2 -> (a -> 处理 iface b st2 st3) ->  
        处理 iface b st1 st3
```

最后,您需要为细化的流程定义更新run和runProc 。清单 15.15 显示了运行所需的更改。您只需修改 Request和Respond的案例,以明确说明进程期望接收的消息类型。

清单 15.15 更新精炼流程(ProcessIFace.idr)的运行

将 service_iface 纳入范围,以便您可以计算预
期的响应类型

根据您发送的消息计算预期的响应类
型

```
run : Fuel -> Process iface t in_state out_state -> IO (Maybe t) run fuel (Request {service_iface} (MkMessage process) msg)  
= do Just chan <- connect process | _ => pure Nothing ok <- unsafeSend chan msg if ok then do Just x <- unsafeRecv  
(service_iface msg) chan
```

| Nothing => pure Nothing pure (Just x) else
pure Nothing run fuel (Respond {reqType} f)

您期望收到一
条满足接
口的消息。

将 reqType 带入范围,以便
您可以说它是接收到的消息的
预期类型

```
= 只是发件人 <- 听 1  
    |什么都没有=>纯 (只是什么都没有)  
只是 msg <- unsafeRecv reqType 发件人  
    |什么都没有=>纯 (只是什么都没有)  
只是 res <- 运行燃料 (f msg)  
    |什么都没有 => 纯粹的没有  
unsafeSend 发件人资源  
纯 (只是 (只是味精))
```

对于runProc,您只需更改其类型即可将iface参数添加到Process:

```
部分 runProc :  
处理 iface () in_state out_state -> IO ()
```

```
runProc proc = 永远运行 proc
    纯的 ()
```

在设计数据类型时,尤其是像Process这样表达强保证的类型时,通常最好先尝试解决特定问题,然后再转向更通用的解决方案。在这里,我们从只支持特定消息和响应类型(Message和Nat)的Process类型开始。只有在那之后,我们才使用类型级函数来创建一个通用的Process类型。

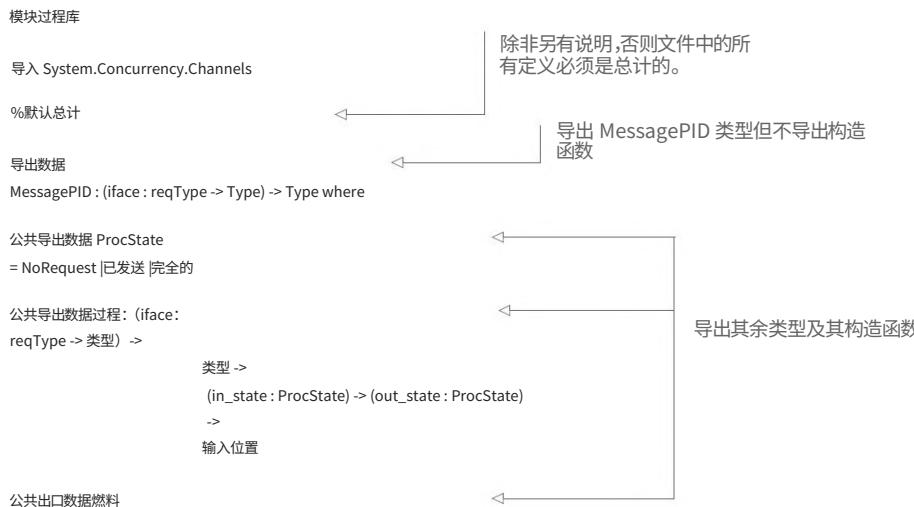
15.2.5 为流程定义一个模块

一旦定义了泛型类型,定义一个新模块以使该类型及其支持功能对其他用户可用是很有用的。我们将定义一个新模块 ProcessLib.idr,它定义Process和支持定义并导出它们

有必要的。

下一个清单显示了模块的整体结构,省略了定义,但在每个声明中添加了导出修饰符。

清单 15.16 在模块中定义流程,省略定义 (ProcessLib.idr)



完整的定义与您已经看到的MessagePID、 ProcState、 Process和Fuel 的定义相同。请记住第 10 章中的数据声明,导出修饰符可以是以下之一:

导出 导出类型构造函数,但不导出数据构造函数。 public export 类型和数据构造函数被导出。

以下清单显示了如何导出支持功能。

清单 15.17 支持Process 的函数类型,省略定义 (ProcessLib.idr)

永远出口部分:燃料

导出运行:
Fuel -> Process iface t in_state out_state -> IO (Maybe t)

公共出口
NoRecv : 无效 -> 类型

公共出口
服务: (iface:reqType -> 类型) -> 类型 -> 类型

公共出口
客户端:类型 -> 类型

export partial runProc :
Process iface () in_state out_state -> IO ()

导出 run 和 runProc 的类型,但不导出它们的定义

导出剩余函数的类型和定义

```

graph TD
    A[永远出口部分:燃料] --> B[导出运行:  
Fuel -> Process iface t in_state out_state -> IO (Maybe t)]
    C[公共出口  
NoRecv : 无效 -> 类型] --> D[服务: (iface:reqType -> 类型) -> 类型 -> 类型]
    E[公共出口  
客户端:类型 -> 类型] --> F[export partial runProc :  
Process iface () in_state out_state -> IO ()]
    B --> G[导出 run  
和 runProc  
的类型,但不  
导出它们的  
定义]
    D --> G
    F --> H[导出剩余函  
数的类型和  
定义]
  
```

对于函数,导出修饰符可以是以下之一:

导出导出类型但不导出定义。 public export 类型和定义被导出。

除非有特定原因要导出定义和类型,否则最好使用导出,隐藏定义的详细信息。在这里,您对Client和Service 使用公共导出,因为它们是类型同义词,其他模块需要知道这些是根据Process 定义的。

现在您已经定义了Process和导出相关定义的单独ProcessLib模块,我们可以尝试更多示例。为了结束本节,我们将看两个使用这种通用Process类型实现并发程序的示例。首先,我们将使用ListType 实现一个进程,它在前面的清单 15.12 中定义,然后我们将看一个更大的例子,它使用并发在后台运行一个进程来计算文件中的字数。

15.2.6 示例1:列表处理

为了演示如何使用Process来定义除procAdder 之外的服务,我们将从响应请求以在List上执行功能的服务开始。服务的接口由ListType函数定义。它提供了两个操作: Length和Append。

```

数据 ListAction : 健入 where
  长度:列表元素 -> ListAction
  追加:列表元素 -> 列表元素 -> 列表动作

ListType : ListAction -> 类型
ListType (长度 xs)= Nat
ListType (Append {elem} xs ys) = List elem
  
```

我们将定义一个procList服务来响应这个接口上的请求。它有以下类型：

```
procList : 服务 ListType ()
```

您可以通过以下步骤逐步定义procList：

1 定义，类型 与procAdder一样，您可以将procList实现为循环，在每次迭代时响应请求：

```
procList : Service ListType () procList = do Respond
(\msg => ?procList_rhs)
    循环过程列表
```

查看这里的?procList_rhs类型，可以看到需要生成的类型是根据收到的msg计算得出的：

```
味精>ListAction
-----
procList_rhs : 处理 ListType (ListType msg) NoRequest NoRequest
```

2 定义 因为您需要生成的类型取决于msg 的值，您可以使用case语句继续定义，检查msg：

```
procList : Service ListType () procList = do Respond
(\msg => case msg of case_val => ?procList_rhs)
```

循环过程列表

case_val上的案例拆分会产生以下结果：

```
procList : Service ListType () procList = do Respond
(\msg => case msg of
    长度 xs => ?procList_rhs_1 附加 xs ys => ?
        procList_rhs_2)
    循环过程列表
```

3 类型，细化对于?procList_rhs_1，如果检查类型，您会发现需要为Length xs的结果生成Nat：

```
msg : ListAction a : Type xs :
List elem
-----
procList_rhs_1 : 处理 ListType Nat NoRequest NoRequest
```

你可以细化?procList_rhs_1如下：

```
procList : Service ListType () procList = do Respond
(\msg => case msg of
    长度 xs => 纯 (长度 xs)
    附加 xs ys => ?procList_rhs_2)
    循环过程列表
```

4 精炼 精炼?procList_rhs_2，需要提供一个List elem，完成定义如下：

```
procList : Service ListType () procList = do Respond
(\msg => case msg of
  length xs => pure (length xs)
  append xs ys => pure (xs ++ ys)
```

完成procList后,您可以通过在进程中生成它并发送请求来尝试它。下面的清单定义了一个向procList实例发送两个请求并显示其结果的过程。

清单 15.18 一个使用procList服务的主程序（ListProc.idr）

```

sequenceDiagram
    participant procMain as procMain : Client()
    participant procList as procList
    participant Nat as Nat

    procMain->>procList: do Just list <- Spawn procList
    activate procList
    procList-->>Nat: | Nothing => Action (putStrLn "Spawn failed") len <- 请求列表 (长度
    activate Nat
    Nat->>procList: [1,2,3])
    activate procList
    procList-->>Nat: 动作 (printLn len)
    activate Nat
    Nat-->>procList: app <- 请求列表 (追加 [1,2,3] [4,5,6])
    activate procList
    procList-->>Nat: 操作 (printLn 应用程序)
    activate Nat

```

设置 procList 进程

调用
长度命令,返回一个

Nat

你可以在REPL上尝试如下：

```
*ListProc> :exec runProc procMain  
3  
[1,2,3,4,5,6]
```

与procAdder一样，procList循环等待传入的请求，并根据需要处理它们，但它在等待请求时不进行任何其他计算。如果并发进程不再花费时间空闲和等待来自其他进程的请求，而是进行一些计算，它们就会变得更加有用。在下一个示例中，您将看到如何执行此操作。

15.2.7 示例2:字数统计过程

当您定义服务时,您可以定义单独的请求来启动操作并获取该操作的结果。例如,如果您要定义字数统计服务,您可以允许客户采取以下步骤:

- 1向字数统计服务发送请求以加载文件并计算字数
文件中的单词。
 - 2当字数统计服务正在处理时,客户端继续自己的工作
例如读取输入和产生输出。
 - 3向字数统计服务发送第二个请求,询问其中有多少字
文件。

在本例中,您将围绕清单 15.19 中定义的WCData记录和doCount函数定义字数统计服务。此函数以字符串的形式获取文件的内容,并生成一个结构,其中包含其中的单词数和行数。

內容

**清单 15.19 一个计算字符串中单词和行数的小函数
(字数.idr)**

```
导入过程库
在哪里记录 WCData
构造函数 MkWCData
字数:自然
线数:自然

doCount : (content : String) -> WCData doCount content = let lcount =
length (lines content) wcount = length (words content) in
MkWCData lcount wcount
```

导入它,以便您可以创建并发的字数统计过
程

有关记录的更多信息,请参见第 12 章。此记录包含文件中单词和行
数的字段。

返回包含内容中单词
和行数的结构

你可以在REPL看到一个这样的例子：

```
*WordCount> doCount  test test\ntest
MkWCData23:WCData
```

目标是实现一个将字数统计作为服务提供的流程。您可以提供两个命令,而不是在单个请求中加载和计算
单词:

CountFile 给定一个文件名,加载该文件并计算字数
在里面

GetData 给定一个文件名,返回该文件的WCData结构,只要该文件已经用CountFile
处理过

CountFile不会返回WCData结构本身,而是立即返回并继续在单独的进程中加载文件。也就是说,一个请
求开始任务,另一个请求检索结果。这将允许请求者在字数统计服务处理文件时继续其自己的工作。以下清
单显示了字数统计服务的接口和框架定义。

清单 15.20 字数统计服务接口 (WordCount.idr)

```
数据 WC = CountFile 字符串
    | 获取数据字符串
WCType : WC -> 类型
WCType (CountFile x) = ()
WCType (GetData x) = 也许 WCData

wcService : (加载 : List (String, WCData)) ->
    | 服务 WCType () wcService 加载
    | wcService 无限循环,并将加载
        | 的文件列表作为参数。
= ?wcService_rhs
```

Returns () 因为它只是
启动一个文件的处理

返回 Maybe WCData 因为
如果给定文件尚未处理,它
将失败

稍后我们将讨论wcService的定义。下一个清单显示了如何调用它并在wcService在后台处理文件时继续
在前台执行交互操作。

定义安全消息传递的类型

清单 15.21 使用字数统计服务 (WordCount.idr)

```

procMain : Client () procMain = do
  Just wc <- Spawn (wcService [])
    |什么都没有 => 操作 (putStrLn "生成失败")
    启动字数统计服
    務的新进程
      动作 (putStrLn "计数 test.txt")
      请求 wc (CountFile test.txt)
        |启动文件的字数统计
      动作 (putStrLn "处理")
      只是 wcdata <- 请求 wc (GetData test.txt)
        |无 => 操作 (putStrLn "文件错误")
        行动 (putStrLn ("字: " ++ 显示 (wordCount wcdata)) ++ 显示 (lineCount wcdata))
        动作 (putStrLn ("行: "))

    在字数统计过程运行时是否有一些工作
    |获取文件字数统计结果
  
```

清单 15.22 展示了wcService的不完整实现,展示了它如何响应命令CountFile和GetData。缺少定义的两部分: 加载和处理请求的文件循环,将新文件信息添加到输入,加载

清单 15.22 响应wcService中的命令 (WordCount.idr)

```

wcService : (loaded : List (String, WCData)) -> Service WCType () wcService 已加载

= 做响应 (\msg => case msg of
  CountFile fname => Pure ()
    |获取数据 fname =>
      纯 (已加载查找 fname)
    ?wcService_rhs
  立即返回 () 以便请求者可以继续处理
  |在现有的已处理文件列表中查找字数数据
  
```

要处理输入,您可以查看Respond 的返回值。请记住,响应具有以下类型:

```

回复 : ((msg : reqType) ->
  处理 iface (iface msg) NoRequest NoRequest) ->
  Process iface (Maybe reqType) st Sent
  
```

Respond的返回值,类型为Maybe reqType,告诉您收到了哪条消息(如果有)。如果wcService收到CountFile命令,它可以在处理下一个输入之前加载和处理必要的文件。

下一个清单显示了wcService 的进一步改进,仍然包括一个用于处理文件的函数。

清单 15.23 wcService 的不完整实现 (WordCount.idr)

```
wcService : List (String, WCData) -> Service WCType () wcService 加载
= do msg <- Respond (\msg => case msg of
    CountFile fname => Pure ()
        获取数据 fname =>
            纯 (已加载查找 fname) )

    newLoaded <- case msg of
        只是 (CountFile fname) =>
            ?countFile 加载的 fname
            => 纯加载
        循环 (wcService newLoaded)
    给定收到的 msg,计算加载的文件数据
    的新列表
```

响应返回收到的消息,您现在可以进一步处理。

只是 (CountFile fname) =>
?countFile 加载的 fname
=> 纯加载

继续加载文件的新列表

如果要求处理文件,请在此处处理。我们将很快定义 countFile。

要查看完成wcService 需要做什么,可以检查?countFile 的类型:

```
加载:列表 (字符串,WCData) fname:字符串 msg2:也许 WC
```

```
st2 : 进程状态
一种
-----
countFile:列表 (字符串,WCData) ->
    String -> Process WCType (List (String, WCData)) Sent Sent
```

countFile需要是一个函数,它获取已处理文件数据的当前列表和文件名,然后返回已处理文件数据的更新列表。下一个清单显示了如何使用前面定义的doCount来定义它来处理文件的内容。

清单 15.24 加载文件并计算单词 (WordCount.idr)

```
countFile:列表 (字符串,WCData) ->字符串->
    Process WCType (List (String, WCData)) Sent Sent
countFile 文件 fname =
    do Right content <- Action (readFile fname)
        | Left err => 纯文件
        让 count = doCount 内容
    动作 (putStrLn ("计数完成")
        纯 ((fname,doCount内容) : .文件))

文件处理完成时向控制台打印一条消息
```

读取文件失败,所以不更新文件列表

++ fname))

将新处理的文件的数据添加到文件列表中

更新 COUNTFILE 的孔请记住,在 wcService 中使用countFile之前,您需要定义它。一旦你定义了 count File,不要忘记用对countFile的调用来替换?countFile洞。

现在您已经定义了 `countFile`, 您可以尝试执行 `procMain`, 它会启动 `wcService`, 要求对文件 `test.txt` 中的单词进行计数, 然后显示结果。

您需要创建一个 `test.txt` 文件, 其内容如下:

```
测试一下
测试
测试测试测试
测试
```

您可以在REPL中执行 `procMain`, 如下所示:

```
*WordCount> :exec runProc procMain 计数 test.txt 处理 test.txt 计数完
成字数:4
行数:7
```

使用 `Process`, 您定义了一种类型, 允许您描述并发执行的进程并解释进程如何按照协议安全地相互发送消息: 服务必须在特定接口上响应它接收到的每条消息, 和

继续循环响应。

然后客户端可以使用正确的接口向进程发送消息, 并被确保收到正确类型的回复。

这并不能解决所有可能的并发编程问题, 但是您已经定义了一种封装了一种并发程序行为的类型。如果描述 `Process` 类型检查的函数是全部的, 那么您可以确信它不会死锁并且所有请求都会收到回复。如果您稍后进一步细化 `Process`, 例如允许对进程之间的交互进行更复杂的描述, 您将能够实现更复杂的并发程序模型。

15.3 总结

并发编程涉及多个进程同时执行

真的。

`Idris` 中的进程通过发送消息相互合作。 `System.Concurrency.Channels`

库提供原始但不安全的,

消息传递的操作。

原始操作是不安全的, 因为它们不能保证进程何时发送和接收消息, 或者发送和接收消息的类型之间的对应关系。

您可以定义一个类型来描述安全的消息传递过程, 使用 `Channel` 作为原语来实现。 使用 `Inf`, 可以保证循环进程继续执行 IO

行动。

434

第15章类型安全的并发编程

通过在类型中定义一个状态机,你可以确定一个进程将
在循环的每次迭代中响应消息。
一个进程的类型可以是通用的,并描述了消息的类型
哪个进程会响应。

附录 A 安装 Idris 和 编辑器模式

本附录说明如何安装 Idris。有适用于 Mac 和 Windows 的预构建二进制发行版，或者您可以在任何类 Unix 操作系统上从源代码安装。它还描述了如何在 Atom 文本编辑器中安装 Idris 模式。

Idris 编译器和环境在撰写本文时，可从<http://idris-lang.org/download>获得适用于 Windows 和 Mac OS 的二进制分发版 Idris。这包括编译器和REPL，以及 Prelude、基础库和各种贡献的库，这些库支持各种数据结构、网络和带有副作用的编程。

在接下来的部分中，我将描述如何将 Idris 作为二进制文件或从源代码安装。无论哪种情况，要检查 Idris 是否已成功安装，您可以运行`idris --version`，它会报告安装了哪个 Idris 版本：

```
$ idris --version 1.0
```

苹果系统

最新版本的 Idris 始终以二进制包的形式提供，可从<http://idris-lang.org/pkgs/idris-current.pkg>下载。为了能够编译并运行您的程序，您还需要安装 Xcode，它可以从 App Store 获得。

因此,要安装 Idris,您可以按照以下步骤操作: 1下载 idris-

current.pkg。

2确保从 App Store 安装 Xcode。

3打开 idris-current.pkg 并按照说明进行操作。这将安装 Idris
二进制到 /usr/local/bin/idris。

您还可以通过 Homebrew (<http://brew.sh/>) 安装最新版本:

```
$ brew install idris
```

视窗

Idris 的预构建二进制文件可从<https://github.com/idris-lang/Idris-dev/>获得
wiki/Windows 二进制文件。这些二进制文件包括编译所需的一切
并运行 Idris 程序。

类 Unix 平台,从源代码安装

Idris 是在 Haskell 中实现的,可以从 Haskell 包管理器 Hackage (<http://hackage.haskell.org/>) 获得源代码分发。这是一个不错的选择

如果您使用的操作系统没有可用的二进制版本,或者

如果您需要更多地控制 Idris 的设置方式。

要安装它,请执行以下步骤: 1安装 Haskell

平台,可从www.haskell.org/downloads获得。

2 Haskell 平台包括一个命令行工具cabal,用于从中央 Hackage 存储库安装应用程序和库。假设你在一个

类 Unix 环境,您可以将 Idris 安装到 /usr/local/ 的可执行文件中
bin/idris 使用以下命令:

```
$ 阴谋集团更新
```

```
$ cabal install idris --program-prefix=/usr/local
```

编辑器模式

在本节中,我将描述如何为类型驱动的交互安装 Idris 模式

在本书中使用的 Atom 文本编辑器中进行编辑。我也给你一些
指向 Emacs 和 Vim 的编辑器模式的指针。

原子

Atom 在<http://atom.io/>上可用于所有主要平台。您可以安装
编辑 Idris 程序的扩展如下:

1转到首选项。

2转到安装选项卡。

3搜索语言-idris包。

4应该有一个搜索结果。单击此结果的安装按钮。

附录A安装 Idris 和编辑器模式

5你需要告诉language-idris包 Idris 二进制文件在哪里

安装。为此,首先转到“包”选项卡。

6单击语言-idris框中的设置。

7在设置标题下,输入安装 Idris 二进制文件的路径。

在此页面上,您还可以更改键盘快捷键和其他各种功能
真的。

其他编辑

在撰写本文时,编辑器模式也可用于 Vim 和 Emacs:

Vim扩展 <https://github.com/idris-hackers/idris-vim>

Emacs模式 <https://github.com/idris-hackers/idris-mode>

在每种情况下,都提供了安装、配置和使用说明。

附录 B

交互式编辑命令

在整本书中,我通过以下方式描述了 Idris 程序的交互式构建
在 Atom 中编辑命令。下表总结了这些命令
方便参考。

表 1 Atom 中的交互式编辑命令

捷径	命令	描述
Ctrl-Alt-A	添加定义	为名称下的名称添加骨架定义 光标
Ctrl-Alt-C	案例拆分	将定义拆分为光标下名称的模式匹配子句
Ctrl-Alt-D	文档	显示该名称的文档 光标
Ctrl-Alt-L	提升孔	将一个洞提升到顶层作为一个新的函数声明
Ctrl-Alt-M	匹配	用匹配中间结果的 case 表达式替换空洞
Ctrl-Alt-R	重新加载	重新加载和类型检查当前缓冲区
Ctrl-Alt-S	搜索	搜索满足光标下孔名称类型的表达式
Ctrl-Alt-T	类型检查名称	显示光标下名称的类型
Ctrl-Alt-W	带块插入	在当前行之后添加一个 with 块,包含一个带有额外参数的新模式匹配子句

附录 C REPL 命令

Idris 读取-评估-打印循环(REPL)提供了几个用于评估的命令
检查表达式和类型、编译程序和搜索文档等。我在整本书中介绍了其中的几个；

下表列出了最常用的命令，但有几个
其他可用。有关更多详细信息，请键入：`?在REPL。`

表 1 Idris REPL 命令

命令	论据	描述
<表达式>	没有任何	显示计算表达式的结果。它包含最近评估的结果的变量。
:t	<表达式>	显示表达式的类型。
:全部的	<名称>	显示具有给定名称的函数是否为总计。
:doc	<名称>	显示名称的文档。
:让	<定义>	添加新定义。
:执行	<表达式>	编译并执行表达式。如果没有给出，则编译并执行 main。
:c	<输出文件>	使用入口点 main 编译为可执行文件。
:r	没有任何	重新加载当前模块。
:l	<文件名>	加载一个新文件。
:模块	<模块名称>	导入一个额外的模块以在 REPL 中使用。

表 1 Idris REPL 命令（续）

命令	论据	描述
:printdef	<名称>	显示名称的定义。
:apropos	<单词>	搜索给定单词的函数名称、类型和文档。
:搜索	<类型>	搜索具有给定类型的函数。
:浏览	<命名空间>	显示给定中定义的名称和类型 命名空间。
:q	没有任何	退出 REPL。

附录 D 进一步阅读

本附录列出了其他一些资源,您可以通过这些资源了解有关函数式编程、类型和 Idris 的理论基础的更多信息。资源按主题分组,每个资源都有简短的评论。

[Haskell Idris 中的函数式编程](#)深受 Haskell 的启发,
包括其语法、语言特性和许多标准库。特别是,Idris 接口与 Haskell 类型类密切相关。如果您想更深入地了解 Haskell,可以查看以下书籍:

Will Kurt 的[Learn Haskell](#) (Manning,2017 年)涵盖了 Haskell 和函数式编程概念,包括几个实际示例,特别强调了 Haskell 的类型系统。[Haskell 编程,第 2 版](#)。作者 Graham Hutton (剑桥大学出版社,2016 年)介绍了 Haskell 的核心特性和纯函数式编程,并且涵盖了 Haskell 的许多更高级的特性。[Richard Bird用 Haskell 进行功能性思考](#) (剑桥大学出版社,2014 年)教授使用 Haskell 从第一原理进行编程,特别强调对程序进行数学推理的技术。

[具有表达类型系统的其他语言和工具](#)由
于使用类型来推理程序正确性的学术研究,
已经出现了几种其他语言。这些是一些例子:

Agda (<http://wiki.portal.chalmers.se/agda/pmwiki.php>) 支持使用依赖类型的类型驱动开发,方式与 Idris 相同,但更强调定理证明。

F* (www.fstar-lang.org/) 一种函数式编程语言,旨在支持使用细化类型进行程序验证,细化类型是用描述该类型值属性的谓词增强的类型。

Coq (<https://coq.inria.fr/>) 一个基于依赖类型的证明管理系统,支持从证明中提取功能程序。

理论基础

Idris 基于对依赖类型理论的数十年研究。您可以从以下来源了解有关理论基础的更多信息:

Simon Thompson 的类型理论和函数式编程 (Addison Wesley, 1991 年)。这本书现已绝版,但可从作者的网站(www.cs.kent.ac.uk/people/staff/sjt/TTFP/) 在线获取。它提供了类型理论的易于理解的介绍,包括其基础和应用。 Benjamin Pierce 等人的软件基础。 (2016 年; www.cis.upenn.edu/~bcpierce/sf/current/index.html) 这是一门关于软件数学基础的课程,从第一原理开始,使用 Coq 重现 Idris 中的示例很有启发性! Benjamin Pierce 的 Types and Programming Languages (麻省理工学院出版社,2002 年)。这本教科书全面介绍了类型系统和编程语言的基本理论。 Philip Wadler 的“作为类型的命题”(Communications of the ACM, 2015 年 12 月; <http://cacm.acm.org/magazines/2015/12/194626-propositions-as-types>)。本文对逻辑和编程语言之间的关系进行了深入但易于理解的说明,特别是编程语言中的类型对应于逻辑命题以及程序对应于该命题的证明的想法。

全函数式编程

在本书中,我讨论了编写全函数的价值,并在第 10 章中介绍了视图,它提供了一种编写全函数的方法。还有一些其他技术,其中一些在以下论文中进行了描述:

David Turner 的“基本强函数式编程”(计算机科学讲义 1022 (1995):1-13)。在这篇论文中,David Turner 主张编写总函数,描述了许多好处,并给出了编写总函数的一些基本技术。 Thorsten Altenkirch、Conor McBride 和 James McKinna 撰写的“为什么依赖类型很重要”(2004 年)。这篇可在线获取的论文(www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf)描述了早期的依赖类型语言 Epigram,强调了整体性在依赖类型编程中的重要性。

“在类型理论中建模一般递归”,Ana Bove 和 Venanzio Capretta (计算机科学中的数学结构15 (2005):671-708)。在这里,作者描述了一种通过定义数据类型来描述程序的计算结构来形式化递归程序的方法。

并发类型第 15 章介绍了一种实

现类型安全并发程序的方法,但它使用了一种非常简化的并发程序形式。您可以通过阅读关于会话类型的更多信息来了解并发编程的类型,由 Kohei Honda 在 1993 年首次描述。有很多来源,但以下论文提供了一个起点: Kohei Honda 的 “Dyadic Interaction 的类型” (讲座笔记)在计算机科学715 (1993):509–523)。本文介绍了会话类型的概念,描述了一种用于对两个进程之间的交互进行编码的类型系统。 Kohei Honda,Nobuko Yoshida 和 Marco Carbone 撰写的 “多方异步会话类型” (编程语言原理, 2008 年)。本文将会话类型的概念扩展到具有两个以上参与者 的通信系统。

Machine Translated by Google

指数

符号

_ (下划线) 66, 83, 160 ::
operator 88, 104,
164 :printdef command
221 : command 166。
function 197 /= method
186, 189 %default total
annotation 312–313 +
operator 225 ++
operator 164, 260 = type
218–219 == operator 183,
185, 209 >>= operator 127–
129 \$ operator 285, 336 \$ =
运营商 346

一个

荒谬的函数 243 累积参数
201
添加命令 168 addCorrect
346 addDownvote 函数
351 加法器函数 157, 406–407

AdderType 函数 155–156 加法函数
155–157 addPositives 339
addToData 函数 166 addToStore 函数
166, 280, 283 addUpvote 函数
351 addWrong 函数 346 allLengths
函数, 细化 65–69 匿名函数的
类型 38–39

追加操作 427 AppendVec.idr
文件 225 使用 205–206 参数为
状态 335–340 定义通用 do 表
示法的应用接口 自动隐式 244–245
定义具有可变数量 155–161 的函数
加法函数 155–157 printf 函数
157 arithInputs 303, 313 算术符号
197 算术测验示例 无限列表
301–304 无限进程 311–313 状态
340–351

Atom 56–64, 436–437 命令
57
数据类型和模式 61–64 通过模式匹配
定义函数 57–61
自动关键字 244 自动隐式概述
245 使用精炼前提条件
388–389

乙

绑定构造函数 320
布尔型 209
布尔值 30
绑定隐式参数 83–84 括号 336 内
置类型 26, 185

C

cabal 工具, Haskell 436 规范
构造函数 216 案例块 177 案例表达式, 在类型 153–154 中使用案例拆分
188, 274, 428 案例语句 210 强制转换函数 28, 198

Cast 接口, 在 198–199 类型之间
转换

包罗万象的案例 190

ATMCmd 类型 387–388

通道库 407–410 字符文字 29–30
 checkEqNat 函数 216–217, 229,
 231, 245
 关闭状态 353
 命令类型 340 注释 47–48
 通信模式 410 复合类型
 40–45 列出函数 43–45 概述 41–42 元
 组 40–41 并发编程 9–10 连接 407–408

连接状态 353
 ConsoleIO 程序 317–318 常量 154
 受约束的泛型类型
 与 Eq 的比较和
 Ord 183–194 约束
 实现 189–191 默认方法定义
 189 定义 Eq 约束 185–189

Ord 191–193 测试
 与 Prelude 194–199 中定义的 Eq
 183–185 接口的相等性

演员表 198–199
 定义数字类型 195–198
 显示 194–195
 参数化的接口
 类型 -> 类型 199–206
 可折叠 201–204
 职能 200–201
 Monad 和 Applicative 205–
 206 受限类型 35–36 控制
 流 132–138 模式匹配绑定 134–136

在交互式定义中生成纯值 132–134
 使用循环编写交互式定义 136–
 138 Control.Monad.State
 325, 333 正确函数 345 CountFile 命令
 430–432 countFrom 函数 294,

296

覆盖函数 262, 277 循环函数 301

D

数据抽象 280–287
 数据存储 282–284 使用视
 图遍历存储内容 284–287
 Data.List.Views 模块 277–
 278 Data.Primitives.Views 311, 348
 数据存储类型精炼 162–164
 使用记录 164–165 死锁 404 Dec 类型
 229–233 decEq 函数 233–234 可判定性
 245–249 Dec 229–233 DecEq 233–
 234 递减参数 266 延迟函数 295, 309
 描述类型规则的依赖状态机 390–402 定义
 抽象游戏状态和操作 391–392 定义具体游
 戏状态 397–399 定义游戏状态 392–395
 的类型 实现游戏 395–397 运行游戏 399–
 402

阅读和验证 138–146

依赖对 141–143 从控制台读取向量
 139–140 读取未知长度的向量 140–
 141 验证向量长度 143–145
 describeHelper 函数 262, 264
 describeListEnd 260–261, 263

除法 303
 做构造函数 306
 do 关键字 179 do
 notation 129–131, 311, 334 为无限进程
 扩展 311 排序命令 320–322 排序表达
 式

可能使用 Monad 键入 177–
 180 和 Applicative 205–206 doAdd
 366 doCount 函数 429 门操作
 示例

操作序列的交互式开发 356–358 建模
 门作为类型 354–356

代表故障 375–378 已验证、错误检查、门
 协议描述 378–382

状态转换中的错误 374–382 类型
 中的安全属性 382–390 定
 义状态 383–384 定义类型 384–387
 使用自动隐式优化前提条件
 388–389 在控制台模拟 387–388
 依赖类型 11–12 定义 102–110
 定义向量 104–107 有界数字的索
 引向量 107–109 车辆分类示
 例 102–103

门关闭状态 353, 358, 375

DoorCmd 类型 355, 374, 378
 By 打开状态 354, 358, 375 byProg 356,
 379
 门结果类型 375
 门状态 374

和

效果库 339
 Elem 谓词 237–250 自动隐式参
 数 244–245 决定向量的成员资
 格 245–249 保证一个值
 在向量 239–241 中删除元素

238–239, 241–244

Emacs模式437空函数
283空类型212,228-229枚举类型,定义89-90

方程约束 182
Eq 接口 211 受约束的实现 189-191
默认方法定义 189
使用接口和实现定义 185-189 测试相等性 183-185

股票常数 186
EqNat 类型 211,239 等式表达式 199 等式类型 可判定性
229-233 年 12 月
DecEq 233-234 空类型 228-229 保证数据的等价性 209-219 = 类型 218-219 精确长度 210-211, 216-218 表示 Nats 的相等性作为类型 211-212 操纵等式 215-216

测试 Nats 212-214 的相等性

相等性推理 220-227 附加向量 225-227 委托证明和重写漏洞 224-225 反转向量 220-221 重写构造 223-224
类型检查和评估 221-222

equalSuffix 函数 279 evalState 330-331
精确长度函数 209-211, 216-218, 233

ExactLength.idr 文件 210
execState 330
出口修饰符 281
Expr 类型 196

F

函数 304
错误值 211
过滤键 286-287
Fin 参数, 使用有界数字 107-109 进行索引
完成类型 391 有限前缀 415 一等类型 22-24
定义具有可变参数数量的函数 155-161 加法函数 155-157

打印函数 157
图式 161-180
数据存储类型 162-165 显示存储中的条目 170-171 孔 165-170
根据 171-175 排序表达式解析条目, 使用 Maybe 使用 do notation 177-180

更新 175-177 类型级函数 148-154 类型同义词 149-150 使用类型 153-154 中的 case 表达式 和模式匹配 150-153

可折叠接口, 使用 201-204 缩小结构
强制函数 295,309 永远函数 310,312,417 格式字符串 148
格式类型 159
来自整数方法 196 的分数接口 195
燃料类型 308, 341 函数
定义 本地 39-40 概述 31-33 函数 30-40, 75-81
匿名 38-39 通过模式
匹配定义 57-61 使用可变数量的参数定义 155-161 加法函数 155-157

格式化输出, 一种类型
安全的 printf 函数 157

高阶 36-38 局部定义 39-40 部分 16-17 部分应用 33 总计 16-17, 296-297, 308-309

type-level 148-154
type synonyms 149-150
using case expressions in types 153-154 with pattern matching 150-153
types of 31 constrained 35-36 Overview 31-33 variables in 33-34

在 84-85 中使用隐式参数编写通用 33-36

函子接口 200,337 在 200-201 的
结构中应用函数
为州 335-340 定义

G

GameLoop 类型 251,391、
399-402
GameState 342-343 泛型类型, 定义 95-100 获取命令 180

获取数据命令 430-431
getEntry 命令 167-168
GetLine 命令 318,341 getPrefix 函数 298-299,308

随机获取 348
GetStr 命令 368
getStringOrInt 151
getValues 函数 287 全局状态 325 greet 函数 315 猜测函数 256 猜谜游戏示例 390-402 定义抽象游戏状态和操作 391-392 定义具体游戏状态 397-399 定义游戏状态类型 392-395

执行 395-397 运行 399-402

H

刽子手猜谜游戏示例 250 完成顶级游戏实现 255 确定输入有效性 255 验证用户输入的谓词 251–252 处理猜测 253–254 表示游戏状态 250–251

顶级游戏功能 251 Haskell 436 headUnequal 234 辅助函数 328 这里价值 240 分层 命名空间 321 高阶函数 36–38 孔 20–21 使用 165–170 委托证明和重写到 224–225 纠正编译错误

自制 436

Idris 编程语言 17–24 作为纯函数式编程语言 13–17 部分和全部功能 16–17 纯度和引用透明性 13–14 副作用程序 14–15

检查类型 18–19 注释 47–48 编译和运行程序 19–20 复合

类型 40–45

具有列表 43–45 的函数 列表 41–42 元组 40–41 编辑器模式 436–437

Atom 436–437 其他编辑器 437 一类

型 22–24 函数 30–40 匿名 38–39 高阶 36–38 局部定义 39–40 部分应用 33

类型和定义 31–33 使用约束类型编写泛型 35–36 使用类型中的变量编写泛型 33–34 孔 20–21 安装编译器和环境 435–436
macOS 435–436
类 Unix 平台 436
视窗 436
互动性 48–50
REPL (读取-评估-打印循环) 17–18 类型 26–30
布尔值 30
字符和字符串 29–30 转换 28 数字 27–28 空格 46
隐式参数 82–85
绑定和未绑定 83–84 需要 82–83 在函数 84–85 中使用
不可能的关键字 229
婴儿型 298
Inf 类型 295–296、308
使用状态机和 418–422 保证响应 应使用 415–418 无限列表 292–305 算术测验示例 301–304 在 293–294 处理中标记 元素 297–299 产生 295–296
流数据类型 299–301 总函数 296–297 无限进程 305–314 算术测验示例 311–313
描述 306–307 执行 307–308 为 311 扩展 do 表示法
使用惰性类型 309–310 生成无限制结构
InfIO 类型 305
中缀运算符 163、336
清单 295
输入和输出处理控制流 132–138 模式匹配绑定 134–136
在交互式定义中产生纯值 132–134
使用循环 136–138 编写交互式定义
IO 通用类型 124–131 >= 运算符 127–129 do notation 129–131 评估和执行交互式程序 125–127 读取和验证依赖类型 138–146
依赖对 141–143 从控制台 139–140 读取向量
读取未知长度的向量 140–141
验证向量长度 143–145
输入功能 272
集成接口 195 交互式数据存储 110–121 在主 113–115 中交互式维护状态
解析用户输入 115–117 处理命令 118–120 表示存储 112–113
交互式操作 329 交互式编程控制流 132–138 读取和验证依赖类型 138–146
IO 通用类型 124–131 接口 声明 Prelude 194–199 中定义的 186 个接口
演员表 198–199
定义数字类型 195–198
显示 194–195
通用比较 183–194
受限实施 189–191

- 接口,通用比较 (续)默认方法定义 189 使用 inter 定义 Eq 匹配 260–261 中的最后一项 反转 264–266 snoc
271–274
- 面孔和实现 185–189 ListType 函数 427 本地可变状态 329 本地状态 325 logOpen
函数 382
- 单词 191–193 循环构造函数 416 loopPrint
测试是否相等 函数 306–307,310 循环,用 136–138 编写交互式定义
- Eq 183–185 由 Type 参数化 -> 类型 199–206
可折叠 201–204
职能 200–201
单子和应用 205–206
-
- IO 操作 301,305,397 MachineCmd 360,362 main
IO 完成 251 函数在 main 113–115 中交互维护
IO 通用类型 124–131 >= 运算符 护状态
- 127–129 do notation 129–131 评估和执行交互式程序
125–127 isInt 153 isList 函数 250
isSuffix 函数 276–277
isValidString 252, 255 it 变量 439 项函数 164 迭代函数 299
概览 318, 406
映射函数 200 个矩阵算术
涉及 6–7 个函数 75–81 操作
和类型 76–77 转置 77–81
-
- 加入方法205
-
- labelFrom 函数 293–294 labelWith 函数 294, 300 语言-idris 包 437
懒惰的注释 277
惰性类型,使用 309–310 生成无限结构
- 长度操作 427 让构造 39–40
- 线性同余发生器 302
- 列出字符 160
列出元素 198 听
407–408
ListLast 261, 272 列出了 41–42 个函数
和 43–45
-
- MkWordState 参数 250 模块,在 Idris 280–282
单子接口
使用 205–206 定义状态 335–340 通
用 do 表示法
- MonadState 接口 330
-
- 匹配 260–261 中的最后一项 反转 264–266 snoc
271–274
- 本地可变状态 329 本地状态 325 logOpen
函数 382
- 树遍历
示例 326–327 状态为 331–332 的树遍历
-
- 第339节
myReverse 函数 222–224,265
myReverseHelper 函数 273
-
- 命名空间 321
纳茨
将相等表示为类型 211–212 概述
156,227,413 测试 212–214
的相等
-
- 否定接口 195 nextState 335
Nil 构造函数 295 非确定性
程序 206 非终止组件 292 noRec 函数 233
-
- NoRecv 接口 424
NoRequest 状态 420
notinNil 248 notinTail 248
未运行状态 391
数字接口 195
numargs 155–156 数字
类型和值 27–28 定义
195–198
-
- 事件函数 184
使用 191–193 定
义排序的 Ord 接口概述 182,267
-
- 对,代表可变状态 328–329 回文函数
279

参数化类型 199 parseBySchema
172 parseCommand 169, 172
parsePrefix 172 部分函数 16–
17 部分函数应用 33

模式匹配扩展视图数据
抽象 280–287 定义和使用
视图 259–270 递归视图 271–279
扩展语法 262–264 具有 150–153
模式匹配绑定的类型级函数
134–136

PID (进程标识符) 406–
407 plus function
221 plusCommutative
type 222 plusSuccRightSucc
227 plusZeroRightNeutral
227
多边形函数 149
位置函数 149–150
谓词
Elem 谓词 237–250 自动隐式参
数 244–245
决定向量 245–249 的成员资格
保证一个值在向量 239–241
中删除元素

238–239, 241–244
刽子手猜谜游戏示例 250 完成顶
级游戏实现 255 确定输入
有效性 255 验证用户输入的谓词
251–252 处理猜测 253–254

代表游戏状态 250–251 顶级游戏
功能 251

原语, 用于并发编程 404–411

通道库 407–410 定义并发进程
406–407 类型错误和阻塞 410–
411

printf 函数 148, 155, 157 PrintfType
159 println 195, 344 问题域 317
procAdder 415–416 进程标识符。参
见 PID processInput 176 ProcessLib
模块 427 procMain 414, 421
ProcState 426 public export 281 纯
构造函数 320, 354 纯函数 205 纯函数
式编程语言 13–17 部分和全部函数
16–17

纯度 13–14 参考
透明度 13–14 副作用程序 14–15

另见 Idris 编程语言纯度 13–14

PutStr 命令 318, 341, 368

问

退出命令 316 测验功能
301–302, 313, 319, 340

R

竞争条件 404–405 随机函数
302, 313
读取操作 405 readGuess
函数 252 readInput 320, 322
记录声明 165 递归 296 递归类
型, 定义 92–95 递归视图
271–279

Data.List.Views 模块 277–
278 与块嵌套 275–277
snoc 列表 271–274 与构造 274–
275 引用透明度 13–14 Refl (自反)
218 removeElem 函数 238–239 重
复函数 299 REPL (读取–评估–打印循环)
17–18, 439

复制功能 314
响应命令 414 返回值 377 反向
函数 265 重写构造 223–224,
235, 273 rewriteCons 274
rewriteNil 274

RingBell 操作 362 运行函数
306–307, 310, 362, 382, 411
runCommand 342, 399

RunIO 类型 315
运行状态 391 runProc
421 runState 函数 330,
333–334, 341, 351, 366

小号

相同的功能 214–215, 220 模式 161–
180
使用孔 165–170 纠正编译错误

使用 164–165 的记
录来细化 162–164 的
数据存储类型
在商店中显示条目 170–171 概述
148 根据 171–175 排序表
达式解析条目

也许使用 do
notation 177–180
更新 175–177
SchemaType 函数 163, 169
发送状态 421
设置难度 345
设置架构命令 176
形状类型 194 显示
功能 194
显示界面, 转换为
带有 194–195 的字符
串 showPrec 函数 194
ShowState 394 副
应用程序 14–15 大小函数 164 排序函数
193

排序向量 70–75 spawn
407–408
拆分列表 267
分裂尼罗河 269
SplitOne 269

- SplitRec** 278
SplitRecPair 277 基于堆栈的计算器示例 367-371
- StackCmd** 364, 368 堆栈 363-371 使用 Vect 366-367 实现 表示操作 364-365 基于堆栈的计算器 例 367-371 状态 算术测验 340-351 定义嵌套记录 343-344 执行 348-350 执行 346-348 更新记录字段值 344-346 通过应用函数更新记录字段 346 自定义实现 333-340 定义 Functor、Applicative 和 Monad 接口 335-340 定义 State 和 runState 333 可变状态 325-332 使用对 328-329 表示 状态类型 329-331 树遍历 例 326-327 树遍历, 状态 331-332 状态机依赖于描述类型 390-402 中的规则
- 响应** 363-371 表示操作 364-365 基于堆栈的计算器 例 367-371 使用 Vect 366-367
- 类型** 353-363 中的跟踪状态 门操作示例 354-358 自动售货机示例 358, 360-362 国家 329 型 状态类型 329-331 定义 333 使用 331-332 storeView 函数进行树遍历 280 StoreView 视图 284 流数据类型 299-301 流类型 293 个流, 无限 列表 292-305 算术测验示例 301-304 标记元素 293-294 处理 297-299 产生 295-296 流数据类型 299-301 总函数 296-297 字符串文字 29-30 StringOrInt 151, 154 strToInt 370 sucNotZero 函数 232 系统模块 311 System.Concurrency.Channels 模块 404, 407
- 吨**
-
- tailUnequal 234 take 函数 299 takeN 函数 270 终止 314-323 域特定命令 317-319 使用 do 表示法的排序命令 320-322
- 布尔值** 30 计算 148-154 类型同义词 149-150 具有模式匹配的类型函数 150-153 在类型 153-154 中使用 case 表达式 字符和字符串 29-30 检查 18-19 检查和评估 221-222 复合 40-45 列表 41-45 元组 40-41 使用 Eq 和 Ord 183-194 接口 定义在 前奏曲 194-199
- Tree data type 203 Tree elem 191 treeLabel function 327 treeLabelWith 328-329 树遍历示例概述 326-327 with State 331-332 修剪功能 152 真值 211 尝试命令 362 元组 40-41 TupleVect type 161 类型 -> 类型参数化接口 199-206 使用 Functor 200-201 通用 do 表示法跨结构应用函数 单子和应用 205-206 减少结构使用 可折叠 201-204 类型同义词 148-150 类型检查 277 类型驱动开发 5-13 自动柜员机 示例 7-9 并发编程 9-10 依赖类型 11-12 矩阵, 算术涉及 10-11 类型的 6-7 过程, 定义 4-5 类型级函数 147 类型 26-30

类型,受约束的通用 (续) 接口由 Type -> Type 参数化 199-206 转换 28 定义 4-5 依赖 11-12 描述规则 390-402 定义抽象游戏状态和操作 391-392 定义具体游戏状态 397-399 定义游戏状态 392-395 的类型实现游戏 395-397
运行游戏 399-402 空 228-229 平等 可判定性 229-234 空类型 228-229 保证 数据等价 209-219 关于平等的推理 220-227 一流 22-24 定义具有可变数量参数的函数 155-161 模式 161-180

类型级函数 148-154 函数 31-33 约束 35-36 变量在 Atom 中 33-34 61-64 数字 27-28, 195-198 安全属性 在 382-390 定义状态 383-384 定义类型 384-387 精炼使用自动隐含的前提条件 388-389 在控制台模拟 387-388 跟踪 353-363 中的状态

门操作示例 354-358 自动售货机示例 358, 360-362 用户自定义 定义 88-101

定义依赖的 102-110 交互式数据存储 110-121 类型安全的并发编程 404-411 的原语

通道库 407-410 定义并发进程 406-407 类型错误和阻塞 410-411 类型用于安全消息传递 411-433

定义模块 426-427 描述流程 412-415 通用流程 422-426 使用状态机和 Inf 保证响应 418-422 列表处理示例 427-429 使用 Inf 使流程总计 415-418 字数统计流程示例 429-433

在

未绑定的隐式参数 83-84 下划线 (_) 66, 83, 160

无人接口 243 个联合类型, 定义 90-92 unsafeReceive 408-410 unsafeSend 408, 410 updateGameState 函数 350 个 用户定义的数据类型, 定义 88-101 枚举类型 89-90 泛型类型 95-100 递归类型 92-95 联合类型 90-92 定义从属 102-110 定义向量 104-107 索引向量与有界数字 107-109 车辆分类

示例 102-103 交互式数据存储 110-121 交互式维护主状态 113-115

解析用户输入 115-117 处理命令 118-120 表示存储 112-113

在

ValidInput 谓词决定输入有效性 255 概述 251-252

valToString 152-153 Vect 大小字符串 162 Vect 类型, 使用 366-367 向量实现堆栈 64-75 附加 225-227 自动精炼 69-70 决定 245-249 的成员

定义 104-107 保证值 在 239-241 索引中, 使用从控制台 139-140 读取的 Fin 107-109 有界编号

长度未知 140-141

allLengths 函数的精炼类型 65-69 从 238-239, 241-244 中删除元素 反转 220-221 排序 70-75 验证长度 143-145

自动售货机示例 造型自动售货机 358

验证描述 360-362 视图 数据抽象 280-287 数据存储 282-284 Idris 中的模块 280-282 遍历存储内容 284-287 定义和使用 259-270 构建视图 262 匹配列表中的最后一项 260-261 合并排序 266-270 使用块 262-264 递归 271-279 反转列表 264-266

递归视图（续）

Data.List.Views 模块 277–278
嵌套块 275–277 snoc 列表 271–274 与构造 274–275
Vim 扩展 437 VList 视图
279 Void 类型空类型 228–229
概述 228, 424

在

WCData 430
wcService 430–431 当函数 335 where 构造 39–40
带有嵌套块的空格 46 275–277 概述 263 扩展模式匹配的语法 262–264

结构 260, 274–275

WordState 类型 250–251, 390
写操作 405 错误函数 345

从

zeroNotSuc 函数 232 zipInputs 211

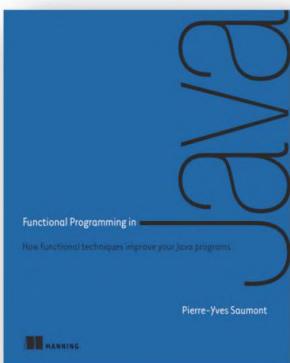
来自曼宁的更多头衔



Paul Chiusano 和 Rúnar Bjarnason
的 Scala 函数式编程

ISBN: 9781617290657

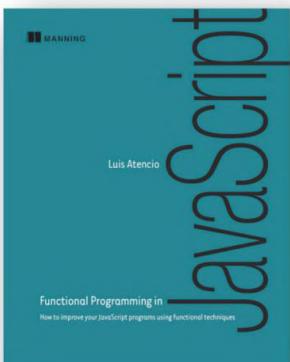
320 页 44.99 美元 2014
年 9 月



Java 中的函数式编程
函数式技术如何改进您的 Java 程序作者
Pierre-Yves Saumont

ISBN: 9781617292736

472 页 49.99 美元 2017
年 1 月

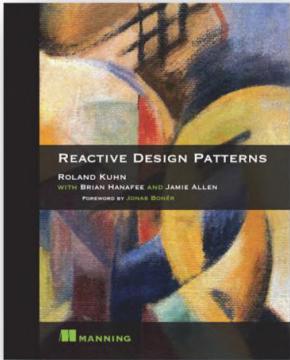


JavaScript 中的函数式编程
Luis Atencio 如何使用函数式技术改进你的 JavaScript 程序

ISBN: 9781617292828

272 页 44.99 美元 2016
年 6 月

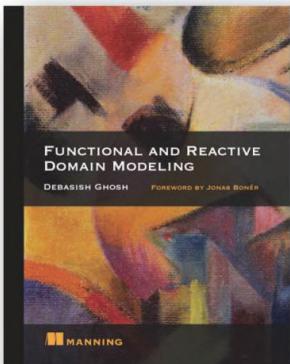
来自曼宁的更多头衔



Roland Kuhn 与 Brian
Hanafee 和 Jamie Allen
的反应式设计模式

ISBN: 9781617291807

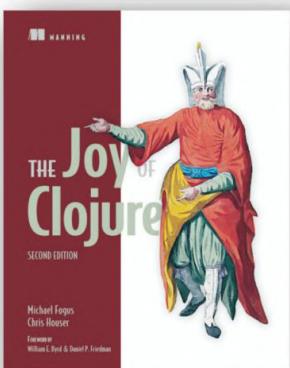
392 页 49.99 美元 2017
年 2 月



Debasish Ghosh 的功能性和反应性领域建模

ISBN: 9781617292248

320 页 59.99 美元 2016
年 10 月



Clojure 的乐趣,第二版,Michael
Fogus 和 Chris Houser

ISBN: 9781617291418

520 页 49.99 美元 2014
年 5 月

如需订购信息,请访问 www.manning.com

来自曼宁的更多头衔

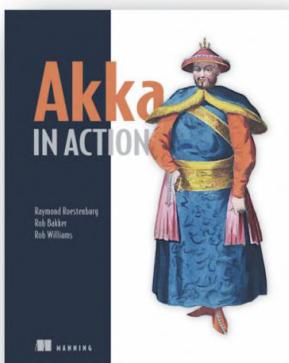


Java 8 在行动

Raoul-Gabriel Urma、Mario Fusco 和 Alan Mycroft 的Lambda、流和函数式编程

ISBN: 9781617291999

424 页 49.99 美元 2014
年 8 月

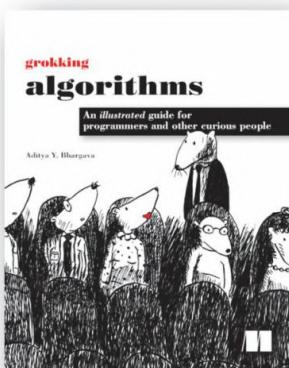


Raymond

Roestenburg、Rob Bakker 和 Rob Williams 的 Akka in Action

ISBN: 9781617291012

448 页 49.99 美元 2016
年 9 月



Grokking 算法

Aditya Y. Bhargava 为程序员和其他好奇的人提供的插图
指南

ISBN: 9781617292231

256 页 44.99 美元 2016
年 5 月

下面详细介绍了用于交互式构建本书中介绍的 Idris 项目的 Atom 命令。

捷径	命令	描述
Ctrl-Alt-A	添加定义	为名称下的名称添加骨架定义 光标
Ctrl-Alt-C	案例拆分	将定义拆分为光标下名称的模式匹配子句
Ctrl-Alt-D	文档	显示该名称的文档 光标
Ctrl-Alt-L	提升孔	将一个洞提升到顶层作为一个新的函数声明
Ctrl-Alt-M	匹配	用匹配中间结果的 case 表达式替换空洞
Ctrl-Alt-R	重新加载	重新加载和类型检查当前缓冲区
Ctrl-Alt-S	搜索	搜索满足光标下孔名称类型的表达式
Ctrl-Alt-T	类型检查名称	显示光标下名称的类型
Ctrl-Alt-W	带块插入	在当前行之后添加一个 with 块,包含一个带有额外参数的新模式匹配子句

软件开发

使用 Idris 进行类型驱动开发

埃德温·布雷迪

停止将类型错误嵌入到你的代码中。这是一种为你的内置文档。

编译器可以用来检查数据关系和其他假设。使用这种方法，您可以在开发早期定义规范并编写易于维护、测试和扩展的代码。Idris 是一种类似于 Haskell 的语言，具有第一类依赖类型，非常适合学习可以应用于任何代码库的类型驱动编程技术。

Idris 的类型驱动开发教您如何利用最先进的类型系统来提高代码的性能和准确性。在本书中，您将学习现实世界软件的类型驱动开发，以及如何处理副作用、交互、状态和并发性。最后，您将能够在 Idris 中开发健壮且经过验证的软件，并将类型驱动的开发方法应用于其他语言。

里面有什么

- 了解依赖类型
- 类型作为一等语言结构
- 类型作为程序构建的指南
- 表达数据之间的关系

为具有函数式编程概念知识的程序员编写。

Edwin Brady 领导 Idris 语言的设计和实现。

要下载 PDF、ePub 和 Kindle 格式的免费电子书，本书的所有者应访问 manning.com/books/type-driven-development-with-idris



“这本书将颠覆你对软件的态度，

以最好的方式。”

伊恩·迪斯，《新遗物》

“强烈推荐给任何认真开发软件的人”
安全要求。

Arnaud Bailly, GorillaSpace

“读完这本书，

对我而言的意义。

Giovanni Ruggiero, Eligotech

“类型驱动开发的清晰完整视图，揭示了力量

”
依赖类型。

尼古拉斯一号

卢森堡科技学院

ISBN-13: 978-1-61729-302-3
ISBN-10: 1-61729-302-4



5 4 9 9 9

9 7 8 1 6 1 7 2 9 3 0 2 3



49.99 美元 / 65.99 美元罐头 [包括电子书]