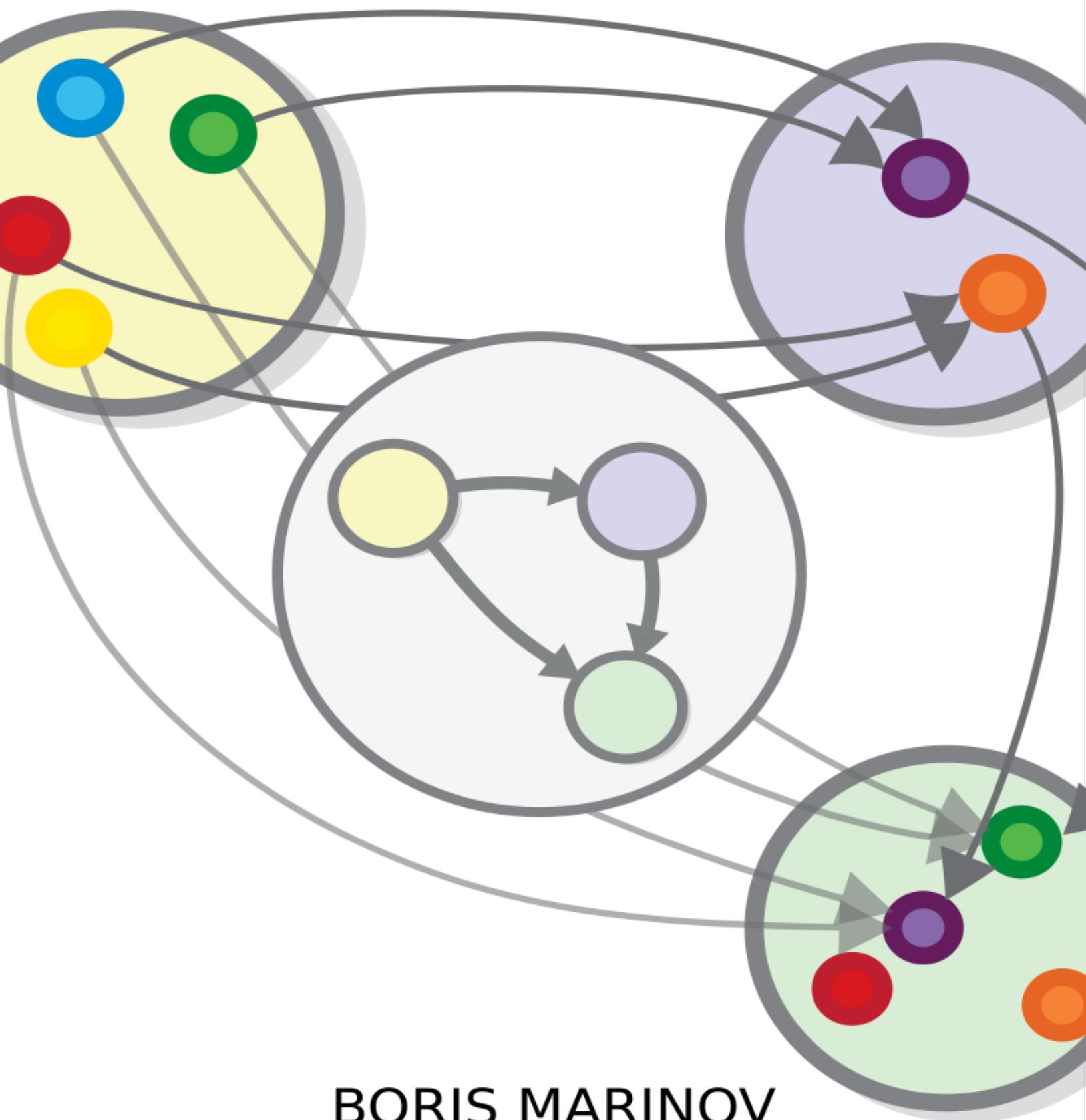
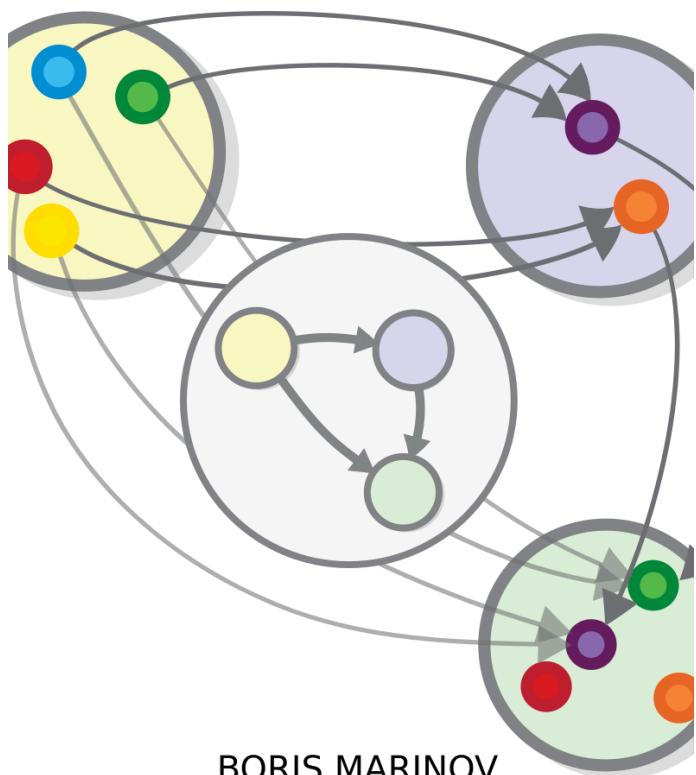


CATEGORY THEORY ILLUSTRATED



BORIS MARINOV

CATEGORY THEORY ILLUSTRATED



BORIS MARINOV

Category Theory Illustrated

Jencel Panic

Creative Commons Non-Commercial Share Alike 3.0

Category Theory Illustrated

[Category Theory Illustrated](#)

[这本书背后的故事](#)

[关于数学](#)

[这本书是为谁写的](#)

[关于范畴论](#)

[总结](#)

[致谢](#)

[集合 \(Sets\)](#)

[什么是抽象理论 \(What is an Abstract Theory\)](#)

[集合 \(Sets\)](#)

[子集 \(Subsets\)](#)

[单例集合 \(Singleton Sets\)](#)

[空集 \(The Empty Set\)](#)

[函数 \(Functions\)](#)

[不同类型的函数 \(Different types of functions\)](#)

[日常生活中的函数 \(Functions in everyday life\)](#)

[恒等函数 \(The Identity Function\)](#)

[函数与子集 \(Functions and Subsets\)](#)

[函数与空集 \(Functions and the Empty Set\)](#)

[函数与单例集合 \(Functions and Singleton Sets\)](#)

[数字的集合与函数 \(Sets and Functions with numbers\)](#)

[数字集合 \(Number sets\)](#)

[数字函数 \(Number functions\)](#)

[编程中的集合与函数 \(Sets and Functions in Programming\)](#)

[集合与类型 \(Sets and types\)](#)

[函数与方法/子程序 \(Functions and methods/subroutines\)](#)

[纯函数式编程语言 \(Purely-functional programming languages\)](#)

[函数组合 \(Functional Composition\)](#)

[关系的组合 \(Composition of relationships\)](#)

[工程中的组合 \(Composition in engineering\)](#)

[组合与外部图 \(Composition and external diagrams\)](#)

[同构 \(Isomorphism\)](#)

同构与恒等 (Isomorphism and identity)
同构与组合 (Isomorphism and composition)
组合同构 (Composing isomorphisms)
单例集合之间的同构 (Isomorphisms Between Singleton Sets)
等价关系与同构 (Equivalence relations and isomorphisms)
等价关系 (Equivalence relations).
自反性 (Reflexivity).
传递性 (Transitivity).
对称性 (Symmetry).
同构作为等价关系 (Isomorphisms as equivalence relations).
插曲: 同构与数字 (Interlude — numbers as isomorphisms).
附录: 软件开发中的组合案例 (Addendum: The case of composition in software development)
从集合到范畴 (From Sets to Categories).
积 (Products).
 插曲——坐标系 (Interlude — coordinate systems).
 作为对象的积 (Products as Objects).
 使用积定义数值运算 (Using Products to Define Numeric Operations).
 通过集合定义积 (Defining products in terms of sets).
 通过函数定义积 (Defining products in terms of functions).
和积 (Sums).
 用集合表示和 (Defining Sums in Terms of Sets).
 用函数表示和 (Defining Sums in Terms of Functions).
插曲: 范畴对偶 (Interlude: Categorical Duality).
 德摩根对偶 (De Morgan Duality).
使用函数定义集合论的其余部分 (Defining the Rest of Set Theory Using Functions).
 用函数定义集合元素 (Defining Set Elements Using Functions).
 用函数定义单例集合 (Defining the Singleton Set Using Functions).
 用函数定义空集 (Defining the Empty Set Using Functions).
 函数应用 (Functional Application).
 结论 (Conclusion).
范畴论简要定义 (Category Theory — Brief Definition).
 集合 vs 范畴 (Sets Vs Categories).
再次定义范畴 (Defining Categories Again).
 复合 (Composition).

[同一律 \(The Law of Identity\)](#)

[结合律 \(The Law of Associativity\)](#)

[交换图 \(Commuting Diagrams\)](#)

[总结 \(A Summary\)](#)

[附录:为什么范畴是这样的? \(Addendum: Why are Categories Like That?\)](#)

[恒等性和同构 \(Identity and Isomorphisms\)](#)

[结合性和还原论 \(Associativity and Reductionism\)](#)

[交换律 \(Commutativity\)](#)

[结合性 \(Associativity\)](#)

[Monoids 等等 \(Monoids etc\)](#)

[什么是幺半群 \(What are monoids\)](#)

[结合律 \(Associativity\)](#)

[单位元 \(The identity element\)](#)

[基础幺半群 \(Basic monoids\)](#)

[从数构成的幺半群 \(Monoids from numbers\)](#)

[布尔代数中的幺半群 \(Monoids from boolean algebra\)](#)

[幺半群运算的集合论定义 \(Monoid operations in terms of set theory\)](#)

[类似幺半群的其他对象 \(Other monoid-like objects\)](#)

[交换幺半群 \(Commutative \(abelian\) monoids\)](#)

[群 \(Groups\)](#)

[总结 \(Summary\)](#)

[对称群及群分类 \(Symmetry groups and group classifications\)](#)

[旋转群 \(Groups of rotations\)](#)

[循环群/幺半群 \(Cyclic groups/monoids\)](#)

[群同构 \(Group isomorphisms\)](#)

[有限群 \(Finite groups\)](#)

[群/幺半群的积 \(Group/monoid products\)](#)

[循环积群 \(Cyclic product groups\)](#)

[阿贝尔积群 \(Abelian product groups\)](#)

[有限阿贝尔群基本定理 \(Fundamental theorem of Finite Abelian groups\)](#)

[颜色混合幺半群作为积 \(Color-mixing monoid as a product\)](#)

[二面体群 \(Dihedral groups\)](#)

[幺半群/群的范畴化视角 \(Groups/monoids categorically\)](#)

[柯里化 \(Currying\)](#)

幺半群元素作为函数 / 置换 (Monoid elements as functions/ permutations).

幺半群操作作为函数组合 (Monoid operations as functional composition).

凯莱定理 (Cayley's theorem).

插曲: 对称群 (Symmetric groups).

幺半群作为范畴 (Monoids as categories).

群/幺半群的表示 (Group/monoid presentations).

插曲: 自由幺半群 (Free monoids).

顺序 (Orders).

线性顺序 (Linear order).

自反性 (Reflexivity).

传递性 (Transitivity).

反对称性 (Antisymmetry).

完备性 (Totality).

自然数的顺序 (The order of natural numbers).

偏序 (Partial order).

链 (Chains).

最大和最小对象 (Greatest and least objects).

并 (Joins).

交 (Meets).

哈斯图 (Hasse diagrams).

颜色顺序 (Color order).

通过除法排序的数字 (Numbers by division).

包含顺序 (Inclusion order).

顺序同构 (Order isomorphisms).

Birkhoff 表示定理 (Birkhoff's representation theorem).

格 (Lattices).

有界格 (Bounded lattices).

半格和树的插曲 (Interlude — Semilattices and Trees).

插曲: 形式概念分析 (Interlude: Formal concept analysis).

预序 (Preorder).

预序与等价关系 (Preorders and equivalence relations).

地图作为预序 (Maps as preorders).

状态机作为预序 (State machines as preorders).

顺序作为范畴 (Orders as categories).

积与余积 (Products and coproducts)

逻辑 (Logic)

什么是逻辑 (What is logic)

逻辑与数学 (Logic and mathematics)

基本命题 (Primary propositions)

组合命题 (Composing propositions)

基本命题与复合命题的等价性 (The equivalence of primary and composite propositions)

推理规则 (Modus ponens)

必然真理 (Tautologies)

公理模式/推理规则 (Axiom schemas/Rules of inference)

逻辑系统 (Logical systems)

结论 (Conclusion)

经典逻辑：真值功能解释 (Classical logic. The truth-functional interpretation)

否定运算 (The negation operation)

插曲：通过真值表证明结果 (Interlude: Proving results by truth tables)

与和或运算 (The and and or operations)

蕴含运算 (The implies operation)

当且仅当运算 (The if and only if operation)

通过公理/推理规则证明结果 (Proving results by axioms/rules of inference)

直觉主义逻辑 (Intuitionistic logic). BHK解释 (The BHK interpretation)

和和或操作

蕴含操作

当且仅当操作

否定操作

排中律

逻辑作为范畴 (Logics as categories)

柯里-霍华德同构 (The Curry-Howard isomorphism)

笛卡尔闭范畴 (Cartesian closed categories)

逻辑作为序 (Logics as orders)

和和或操作

否定操作

蕴含操作

当且仅当操作

范畴逻辑的初体验 (A taste of categorical logic)

真和假

和和或

蕴含

函子 (Functors)

我们迄今看到的范畴 (Categories we saw so far)

集合范畴 (The category of sets)

特殊类型的范畴 (Special types of categories)

其他范畴 (Other categories)

有限范畴 (Finite categories)

审视一些有限范畴 (Examining some finite categories)

范畴同构 (Categorical isomorphisms)

集合同构 (Set isomorphisms)

序同构 (Order isomorphisms)

范畴同构 (Categorical isomorphisms)

范畴同构的问题 (The problem with categorical isomorphisms)

什么是函子 (What are functors)

对象映射 (Object mapping)

态射映射 (Morphism mapping)

函子定律 (Functor laws)

日常语言中的函子 (Functors in everyday language)

图表是函子 (Diagrams are functors)

地图是函子 (Maps are functors)

人类感知具有函子性质 (Human perception is functorial)

单子中的函子 (Functors in monoids)

对象映射 (Object mapping)

态射映射 (Morphism mapping)

函子定律 (Functor laws)

序中的函子 (Functors in orders)

对象映射 (Object mapping)

态射映射 (Morphism mapping)

函子定律 (Functor laws)

线性函数 (Linear functions)

编程中的函子——列表函子 (Functors in programming. The list functor)

类型映射 (Type mapping)

函数映射 (Function mapping)

函子定律 (Functor laws)

函子的用途 (What are functors for)

指向函子 (Pointed functors)

自函子 (Endofunctors)

恒等函子 (The identity functor)

指向函子 (Pointed functors)

小范畴的范畴 (The category of small categories)

层层范畴 (Categories all the way down)

Category Theory Illustrated

layout: default

title: 关于

{: style="text-align: center"}

纪念

{: style="text-align: center"}

弗朗西斯·威廉·劳维尔 (Francis William Lawvere)

{: style="text-align: center"}

1937 — 2023

{: style="text-align: right"}
“你尽力了，

{: style="text-align: right"}
但你就是无法逃避数学”

{: style="text-align: right"}
汤姆·莱勒 (Tom Lehrer)

这本书背后的故事

我小时候对数学感兴趣，但总是搞错计算，所以我决定这不是我的专长，并开始追求其他兴趣，比如写作和视觉艺术。

后来我开始接触编程，发现它与我喜欢的那部分数学很相似。为了探索这种相似性并提高自己的开发能力，我开始使用函数式编程。不久后，我发现了范畴论 (Category Theory)。

大约五年前，我失业了几个月，决定发布一些我在阅读大卫·斯皮瓦克 (David Spivak) 的《科学家的范畴论 (Category Theory for Scientists)》时记笔记所绘制的图。这一努力产生了本书前两章的粗略版本，我将其发布到网上。

几年后，一些人找到了我的笔记并鼓励我写更多。他们非常友善，以至于让我忘记了自己的冒充者综合症，开始着手写接下来的几章。

关于数学

自从牛顿 (Newton) 的《自然哲学的数学原理 (Principia)》以来, 数学被视为“科学和工程的工作马”, 即数学仅被视为一种帮助科学家和工程师取得技术和科学进展的工具, 仅用于解决“实际”问题。

因此, 数学家处于一个奇怪且独特的位置, 始终需要为他们所做的工作辩护, 证明它对其他学科的价值。我再强调一次, 这种情况在其他学科上显得荒谬。

没有人期望从物理理论中得到回报。例如, 没有人因为某个物理理论没有实用价值而批评它。

批评哲学理论不实用则更为荒唐。试想有人批评维特根斯坦 (Wittgenstein):

“很好, 但语言的图像理论能做什么?” “我听说它在编程语言理论中有些应用……”

或者对大卫·休谟 (David Hume) 的怀疑论表示怀疑:

“很好, 但你的理论让我们在知识上原地踏步。接下来我们该做什么?”

虽然许多人不一定认同数学是工具的观点, 但我们在大多数数学教科书的结构中看到这种编码: 每章以概念解释开始, 然后是一些例子, 最后是列出该概念所解决的问题。

这种方法没有错, 但数学远不止是解决问题的工具。它曾是古希腊宗教教派 (毕达哥拉斯学派) 的基础, 被哲学家视为理解宇宙法则的手段。它曾是, 也仍然是, 一种使不同文化背景的人能够相互理解的语言。它也是一种艺术和娱乐方式。它是一种思维模式, 甚至可以说它就是思维本身。有人说“写作即思考”, 但我认为, 写作一旦足够精炼并且没有任何作者偏见, 就自动成为数学写作——你几乎可以将文字转换为公式和图表。

范畴论 (Category Theory) 体现了数学的所有这些方面, 因此我认为它是写一本展示所有这些方面的书的良好基础——一本不以解决问题为基础, 而是探索概念并寻求它们之间联系的书。一本整体上很美的书。

这本书是谁写的

那么，这本书是为谁写的呢？有些人可能会将这个问题表述为“谁应该读这本书”，但如果你这样问，答案就是“没有人”。的确，如果你以“应该”的角度思考，数学（至少本书涉及的数学类型）不会对你有多大帮助，尽管它被虚假地宣传为解决许多问题的方案（而实际上它是，正如我们所述，远不止如此）。

让我们举个例子——许多人声称爱因斯坦 (Einstein) 的相对论对于全球定位系统 (GPS) 的正常运行至关重要。由于相对论效应，GPS 卫星上的时钟比地面上的相同时钟走得更快。

他们似乎认为，如果没有该理论，开发 GPS 的工程师会面对以下情况：

工程师1：哇，卫星上的时钟快了X纳秒！

工程师2：但这不可能！我们的数学模型预测它们应该是正确的。

工程师1：好的，那我们现在该怎么办？

工程师2：我想我们得搁置这个项目，直到我们有一个描述宇宙时间的数学模型。

虽然我不是相对论 (special relativity) 的专家，但我怀疑这段对话的发展更接近于以下情况：

工程师1：哇，卫星上的时钟快了X纳秒！

工程师2：这是正常的。存在许多未知数。

工程师1：好的，那我们现在该怎么办？

工程师2：只需调整X看看是否有效。哦，顺便告诉物理学家，他们可能会觉得有趣。

换句话说，我们可以在没有任何高级数学甚至没有数学的情况下解决问题，正如埃及人能够在不知道欧几里得几何 (Euclidean geometry) 的情况下建造金字塔一样。而我要说的是，数学不仅仅是一个简单的工具。阅读任何一本数学教科书（当然，尤其是这本书）都会在许多更重要的方面帮助你，而不仅仅是解决“复杂”问题。

有些人说我们在日常生活中不使用数学。但如果这是真的，那只是因为其他人已经为我们解决了所有难题，并且这些解决方案编码在我们使用的工具中。但是，如果你不了解数学，那你将永远是一个消费者，依赖于现有的工具和解决方案，而无法自行解决任何问题。

因此，“这本书是为谁写的”这个问题不应该被理解为谁应该读，而是谁可以读。答案是：“每个人”。

关于范畴论

如我们所述,数学的基础是思维的基础。范畴论 (Category Theory) 让我们能够形式化我们在日常 (智力) 生活中使用的这些基础。

我们的思考和交流方式是基于自然发展起来的直觉,这是我们传达观点的非常简单的方式。然而,直觉也使得误解变得容易——我们所说的话通常可以以多种方式解释,其中有些是错误的。这类误解是偏见产生的原因。此外,有些人(在古希腊被称为“诡辩家”)会故意引入偏见,以便获得短期利益(不关心引入偏见对每个人的长期伤害)。

在这种情况下,人们通常会诉诸公式和图表来完善他们的想法。图表(甚至比公式更常见)在科学和数学中无处不在。

范畴论形式化了图表及其组成部分——箭头 (arrows) 和对象 (objects) ——以创建一种表示各种想法的语言。在这个意义上,范畴论是一种统一知识的方式,既包括数学知识,也包括科学知识,并且通过通用术语统一各种思维模式。

因此,范畴论和图表也是一种非常易于理解的方式来清晰地传达一个正式的概念,我希望在接下来的几页中能展示这一点。

总结

在本书中, 我们将探索各种知识模式, 并在此过程中, 通过范畴的视角 (lens of categories) 看到各种数学对象。

我们从第一章的集合论 (*set theory*) 开始, 这是形式化不同数学概念的最初方式。

第二章我们将进行一个(希望)平滑的过渡, 从集合到范畴 (*categories*), 同时展示它们的比较, 最终引出范畴论的定义。

接下来的两章, 第三章和第四章, 我们将跳入数学的两个不同分支, 并介绍它们的主要抽象手段, 群 (*groups*) 和 序 (*orders*), 观察它们如何与我们之前介绍的核心范畴论概念相联系。

第五章也遵循前两章的主要公式, 深入探讨为什么范畴论是一种通用语言,

展示它与古老学科逻辑 (*logic*) 的联系。与第三章和第四章一样, 我们从逻辑本身的速成课程开始。

第六章我们探讨所有这些不同学科之间的联系, 使用一个最有趣的范畴理论概念——函子 (*functor*) 的概念。

在第七章中, 我们将回顾另一个更有趣且更高级的范畴概念, 自然变换 (*natural transformation*) 的概念。

致谢

感谢我的妻子迪米特丽娜 (Dimitrina), 感谢她的支持。

感谢我的女儿达里亚 (Daria), 我的“反作者”, 她坐在我膝上写第二、三章时无情地删除了许多句子, 其中大部分都是糟糕的句子。

感谢我的高中美术老师乔治耶娃夫人 (Mrs. Georgieva), 她告诉我我有些天赋, 但需要努力。

感谢普拉休什·普拉莫德 (Prathyush Pramod), 他鼓励我完成这本书并帮助了我。

也感谢所有提交反馈并帮助我修正我所犯的无数错误的人——了解我自己, 我知道还有更多。

以下是按您要求的纯Markdown文本翻译:

layout: default

title: 集合 (Sets)

集合 (Sets)

让我们通过研究集合的基本理论开始我们的探讨。集合论 (Set Theory) 和范畴论 (Category Theory) 共享许多相似之处。我们可以将范畴论视为集合论的泛化。也就是说，范畴论旨在描述与集合论相同的事物（所有事物？），但以一种更抽象、更灵活且（希望）更简单的方式进行描述。

换句话说，集合是范畴的一个例子（我们可以称之为原型例子），有例子是很有用的。

什么是抽象理论 (What is an Abstract Theory)

“我们不再问从假设出发可以定义和推导出什么，而是询问可以找到什么更普遍的想法和原则，通过这些想法和原则可以定义或推导出我们的起点。”

—— 伯特兰·罗素 (Bertrand Russell), 《数学哲学导论 (Introduction to Mathematical Philosophy)》

大多数科学和数学理论都有一个特定的领域，它们与该领域紧密相关并且在该领域有效。它们是为该领域创建的，并不打算在其他领域使用。例如，达尔文的进化论是为了解释不同的生物物种如何通过自然选择进化，而量子力学则描述了粒子在特定尺度上的行为等。

即使大多数数学理论虽然并非固有地绑定到某个特定领域（如科学理论那样），它们至少与某个领域有很强的联系，例如微分方程是为描述事物如何随时间变化而创建的。

集合论和范畴论不同，它们不是为了提供关于某个特定现象的严格解释，而是提供了一个更为普遍的框架，用于解释各种现象。它们的作用不像工具，而更像是定义工具的语言。这样的理论被称为**抽象理论**。

有时两者的界限是模糊的。所有理论都使用**抽象**，否则它们将毫无用处：如果没有抽象，达尔文将不得不谈论具体的动物物种甚至单个动物。但理论有一些核心概念，这些概念不指代任何特定事物，而是供人们进行泛化。所有理论都适用于它们的领域之外，但集合论和范畴论一开始就没有特定的领域。

具体理论，如进化论，由具体概念组成。例如，*种群 (population)*，也叫*基因库 (gene-pool)*，指的是可以相互繁殖的个体群体。而抽象理论，如集合论，则由抽象概念组成，例如集合的概念。集合的概念本身不指代任何特定事物。然而，我们不能说它是空洞的概念，因为有无数事物可以用集合表示，例如，基因库可以（非常恰当地）用个体动物的集合来表示。物种

也可以用集合表示——一个由所有理论上可以相互繁殖的种群组成的集合。

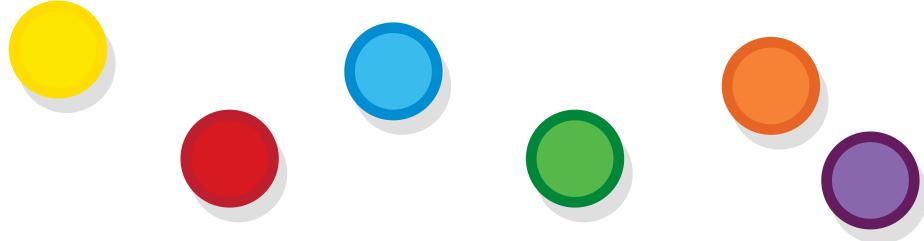
你已经看到了抽象理论的有用之处。因为它们如此简单，可以作为许多具体理论的构建块。因为它们具有普遍性，可以用来统一和比较不同的具体理论，通过将这些理论置于共同的基础上（这是范畴论非常典型的特征，我们稍后会看到）。此外，好的（抽象）理论可以作为思维模型，帮助我们发展思维。

集合 (Sets)

“集合是我们感知或思维的确定的、独特的对象的集合，这些对象被称为集合的元素。”

——格奥尔格·康托尔 (Georg Cantor)

也许毫不奇怪，集合论中的一切都是用集合来定义的。集合是事物的集合，其中的“事物”可以是你想要的任何东西（如个体、种群、基因等）。例如，考虑这些球。



Balls

让我们构建一个集合，称之为 G （灰色 (gray)），它包含所有这些球作为元素。这样的集合只能有一个：因为集合没有结构（没有顺序，没有一个球在另一个球之前或之后，也没有成员在集合中有“特殊”地位）。包含相同元素的两个集合只是同一个集合的两个图片。



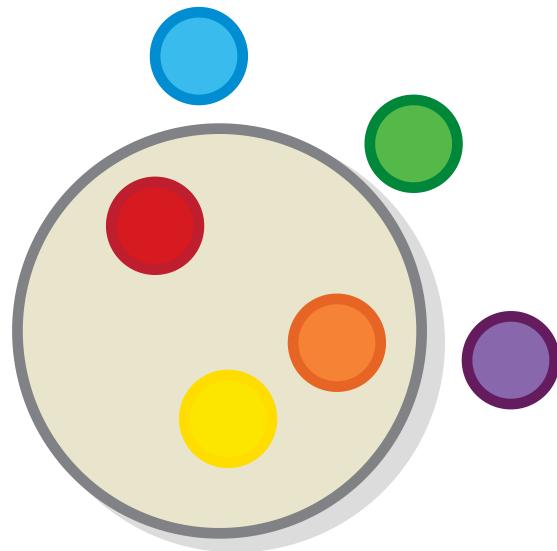
所有球的集合

这个例子看起来可能过于简单，但实际上，它和其他例子一样有效。

使这个概念有用的关键见解在于，它使你能够像处理一个事物一样处理多个事物。

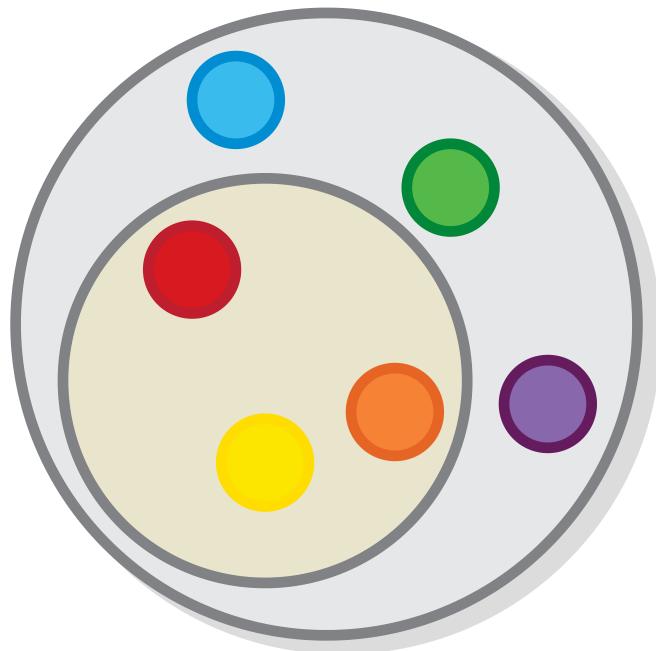
子集 (Subsets)

让我们再构建一个集合。这个集合包含所有颜色温暖的球。我们称之为 Y (因为在图中，它以yellow(黄色)显示)。



所有颜色温暖的球的集合

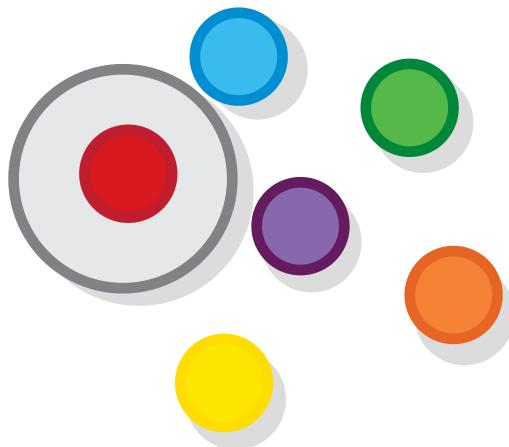
注意, Y 只包含也出现在 G 中的元素。也就是说, 集合 Y 的每个元素也是集合 G 的元素。当两个集合具有这种关系时, 我们可以说 Y 是 G 的一个子集(或 $Y \subseteq G$)。当两个集合一起绘制时, 子集完全位于其超集中。



Y 和 G 一起

单例集合 (Singleton Sets)

所有红球的集合只包含一个球。我们之前说过，集合将多个元素汇总为一个。但即使包含一个元素的集合也是完全有效的——简而言之，有些事物是独一无二的。一个王国的国王/王后的集合就是一个单例集合。

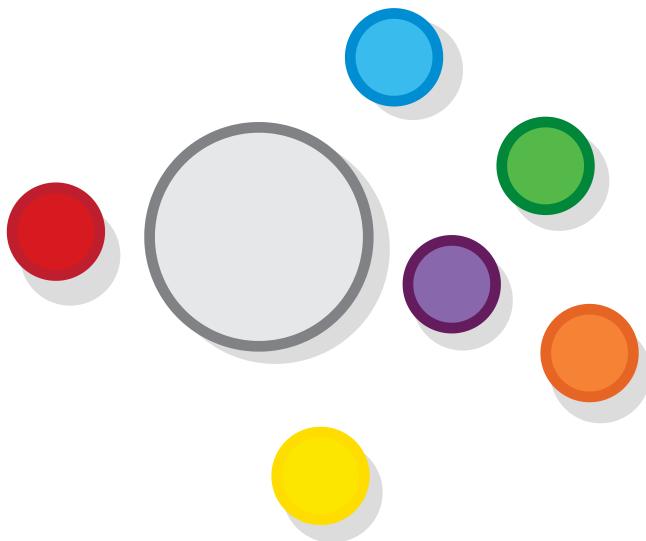


红球的单例集合

单例集合有什么意义呢？嗯，它是集合论语言的一部分，例如，如果我们有一个期望给定项集合的函数，而只有一个项满足条件，我们可以只用该项创建一个单例集合。

空集 (The Empty Set)

当然，如果一个是有效的答案，那么零也可以是。如果我们想要一个所有黑球的集合 B 或所有白球的集合 W ，所有这些问题的答案都是相同的——空集。



空集

因为集合仅由其包含的项目定义, 所以空集是唯一的——例如, 包含零个球的集合和包含零个数字的集合之间没有区别。形式上, 空集用符号 \emptyset 表示(所以 $B = W = \emptyset$)。

空集有一些特殊属性, 例如, 它是每个其他集合的子集。数学上, $\forall A \rightarrow \emptyset \subseteq A$ (\forall 表示“对于所有”)。

函数 (Functions)

“我所说的函数是指将各种表象排列在一个共同表象之下的统一性。”

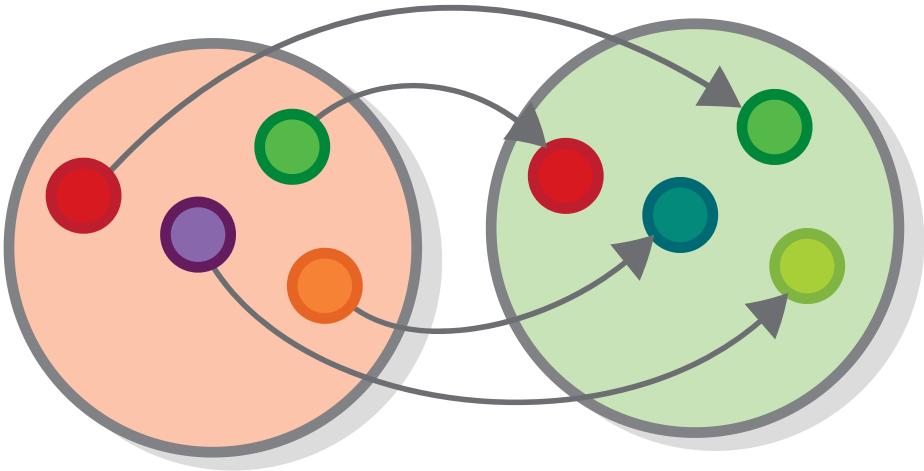
—— 伊曼纽尔·康德 (Immanuel Kant), 《纯粹理性批判 (The Critique of Pure Reason)》

函数是两个集合之间的关系, 它将一个集合的每个元素与另一个集合中的一个元素对应起来, 第一个集合称为函数的源集合, 第二个集合称为函数的目标集合。

这些集合也被称为函数的定义域和值域, 或输入和输出。在编程中, 它们分别被称为参数类型和返回类型。在逻辑中, 它们对应于前提和结论 (我们稍后会讨论)。根据情况不同, 我们也可以说一个给定的函数从这个集合到另一个集合, 连接这两个集合, 或者它将一个集合的值转换为另一个集合的值。这些不同的术语表明了函数概念的多面性。

不同类型的函数 (Different types of functions)

这是一个函数 f , 它将集合 R 中的每个球转换为另一个集合 G 中颜色相反的球 (在数学中, 函数的名称通常伴随着其源集合和目标集合的名称, 如下所示: $f: R \rightarrow G$)。



相反的颜色

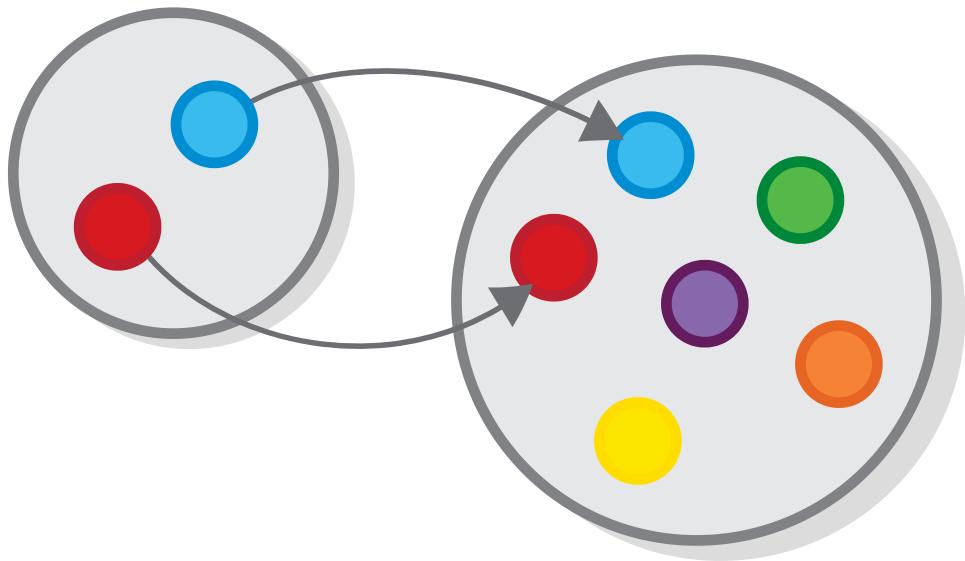
这可能是存在的最简单类型的函数之一——它编码了集合之间的一对一关系。也就是说，源集合中的一个元素被连接到目标集合中的一个元素(反之亦然)。

但函数也可以表达多对一的关系，其中源集合中的多个元素可能被连接到目标集合中的一个元素(但反之则不成立)。下面是一个这样的函数。

![从较大的集合到较小的集合的函数](..//01_set/function_big_small.svg)
)

这样的函数可能代表诸如根据某些标准对给定对象集合进行分类，或根据某种属性对它们进行划分的操作。

函数还可以表达某些目标集合中的元素不参与的关系。



从较小集合到较大集合的函数

一个例子可能是某种模式或结构与这种模式在某个更复杂的上下文中的出现之间的关系。

我们看到了函数的多样性，但有一件事是你在函数中无法拥有的。你不能有一个源元素没有被映射到任何东西，或者被映射到多个目标元素——那将构成一个多对多关系，而我们说过函数表达的是多对一关系。这种“设计决策”是有原因的，我们很快就会谈到它。

日常生活中的函数 (Functions in everyday life)

集合和函数可以表达各种对象，甚至是人之间的关系。你提出的每一个有答案的问题都可以表示为一个函数。

问题“我们离纽约有多远？”是一个以世界上所有地方的集合为源集合，目标集合由所有正数的集合组成的函数。

问题“我父亲是谁？”是一个源为世界上所有人的函数。

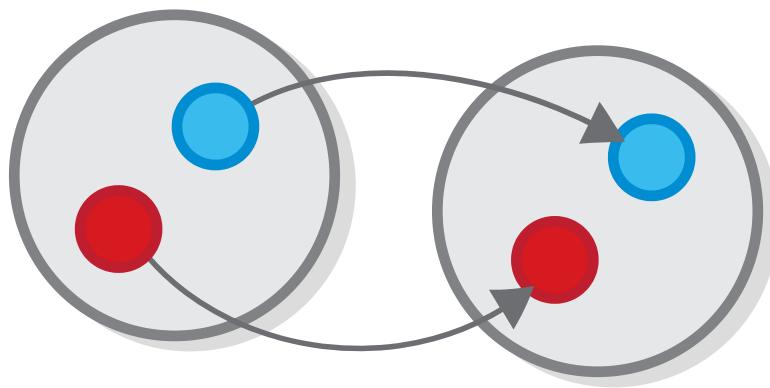
问题：这个函数的目标是什么？

注意，问题“我有孩子吗？”不是一个直接的函数，因为一个人可能没有孩子，或者可能有多个孩子。我们将学习如何将这些问题表示为函数。

问题： 我们一开始画的所有函数是否都表达了某种东西？你认为一个函数需要表达某些东西才能有效吗？

恒等函数 (The Identity Function)

对于每个集合 G ，无论它表示什么，我们都可以定义一个什么都不做的函数，换句话说，一个将 G 中的每个元素映射到自身的函数。它被称为 G 的 **恒等函数**，或 $ID_G : G \rightarrow G$ 。

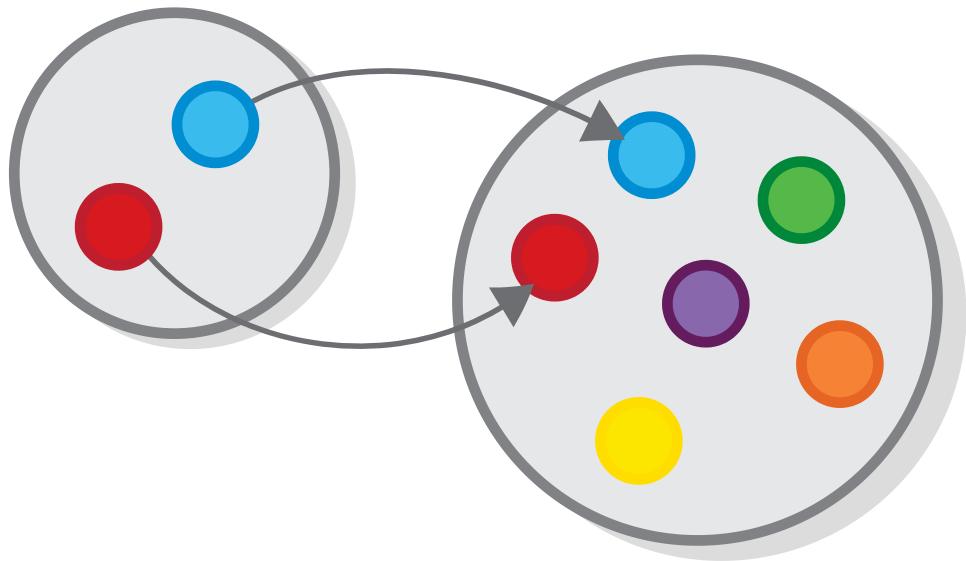


恒等函数

你可以将 ID_G 视为一个代表 G 在函数领域中的函数。它的存在使我们能够正式地证明许多我们“直觉上知道”的定理。

函数与子集 (Functions and Subsets)

对于每个集合和子集，无论它们表示什么，我们都可以定义一个函数（称为子集的像），它将子集中的每个元素映射到自身：

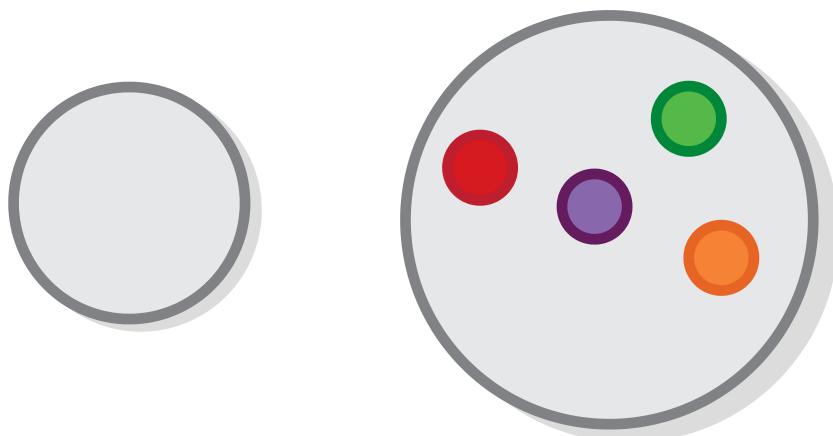


从较小集合到较大集合的函数

每个集合都是它自己的子集,在这种情况下,这个函数与恒等函数相同。

函数与空集 (Functions and the Empty Set)

从空集到任何其他集合都有一个唯一的函数。



带有空集的函数

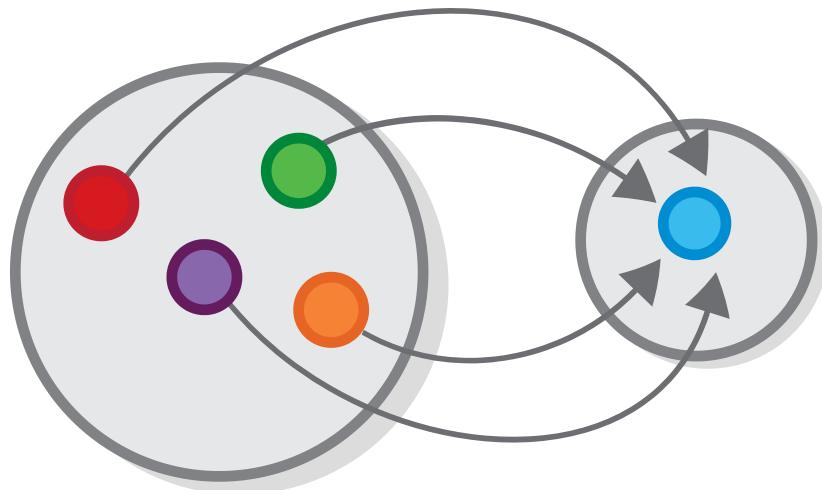
问题: 这真的有效吗?为什么?检查定义。

请注意，这一说法也是从“存在一个子集到集合的函数”的结果，以及“空集是任何其他集合的子集”的结果得出的。

问题：反过来呢？有没有以空集为目标而不是以空集为源的函数？

函数与单例集合 (Functions and Singleton Sets)

从任何集合到任何单例集合都有一个唯一的函数。



带有单例集合的函数

问题：这真的是将任何集合连接到单例集合的唯一有效方法吗？

问题：再次反过来呢？

数字的集合与函数 (Sets and Functions with numbers)

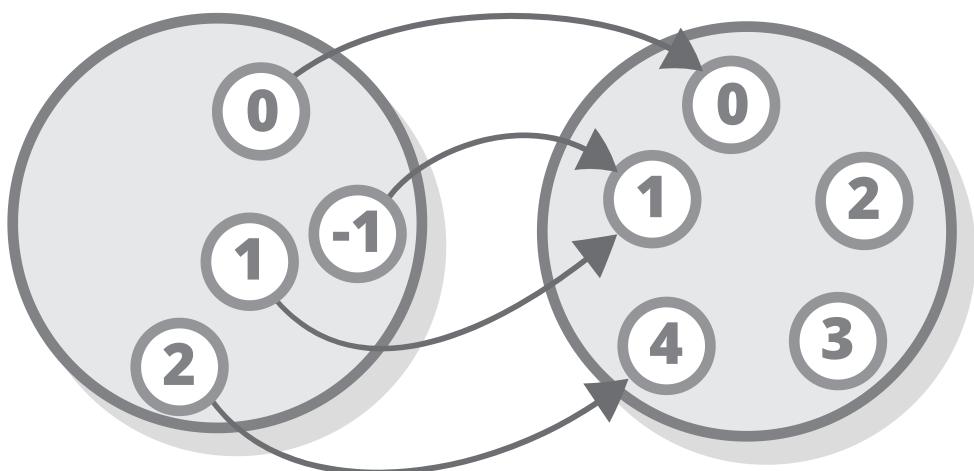
所有的数字运算都可以表示为在不同类型的数字集合上作用的函数。

数字集合 (Number sets)

因为并非所有函数都适用于所有数字, 我们将数字集合划分为几个集合, 其中许多集合是彼此的子集, 例如整数集合 $\mathbb{Z} := \dots -3, -2, -1, 0, 1, 2, 3\dots$, 正整数集合(也称为“自然”数), $\mathbb{N} := 1, 2, 3\dots$ 。我们还有实数集合 \mathbb{R} , 它包括几乎所有数字, 以及正实数集合(或 $\mathbb{R}_{>0}$)。

数字函数 (Number functions)

每个数字运算都是这些集合之间的函数。例如, 平方一个数是从实数集合到非负实数集合的函数(因为这两个集合都是无限的, 我们无法绘制它们的全貌, 但我们可以绘制它们的一部分)。



平方函数

借此机会,我想重申函数的一些重要特征:

- 目标集合中的所有数字都有(或应该有)两条箭头指向它们(一个对应正平方根,另一个对应负平方根),这是可以的。
- 源集合中的零与目标集合中的自身连接——这是允许的。
- 某些数字不是其他数字的平方——这也是允许的。

总体而言,只要每个值都能提供唯一结果,一切都是允许的。对于数字运算,这总是正确的,因为数学的设计就是这样。

“每一个数字的泛化最初都呈现为解决某个简单问题所需的:为了使减法总是可能的,需要负数,否则当 $a < b$ 时 $a - b$ 将没有意义;为了使除法总是可能的,需要分数;为了使提取根和解方程总是可能的,需要复数。”

——伯特兰·罗素 (Bertrand Russell),《数学哲学导论 (Introduction to Mathematical Philosophy)》

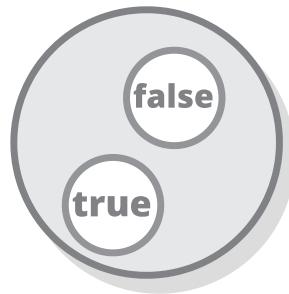
注意,大多数数学运算,如加法、乘法等,都需要两个数字才能产生结果。这并不意味着它们不是函数,只是它们有点复杂。根据需要,我们可以将这些运算表示为从数字元组集合到数字集合的函数,或者我们可以说它们接受一个数字并返回一个函数。稍后我们会详细讨论。

编程中的集合与函数 (Sets and Functions in Programming)

集合在编程中被广泛使用，尤其是在其作为类型(也称为类)的体现中。我们之前讨论的所有数字集合在大多数编程语言中也作为类型存在。

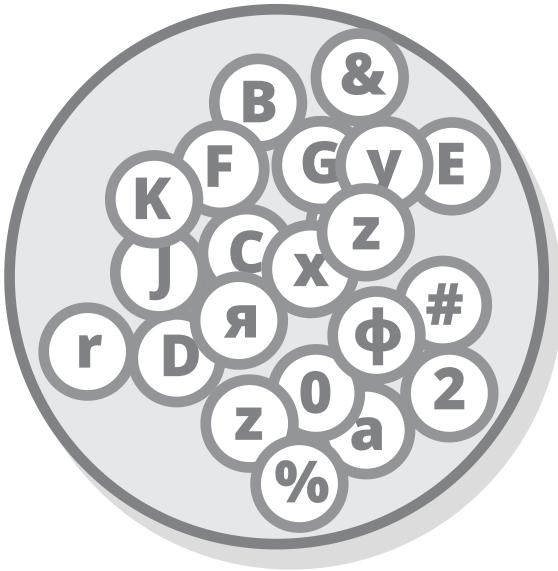
集合与类型 (Sets and types)

集合与类型并不完全相同，但所有类型都是(或可以视为)集合。例如，我们可以将 Boolean 类型视为包含两个元素的集合——true 和 false。



布尔值集合

编程中的另一个非常基本的集合是键盘字符的集合，或 Char。字符实际上很少单独使用，通常作为序列的一部分使用。



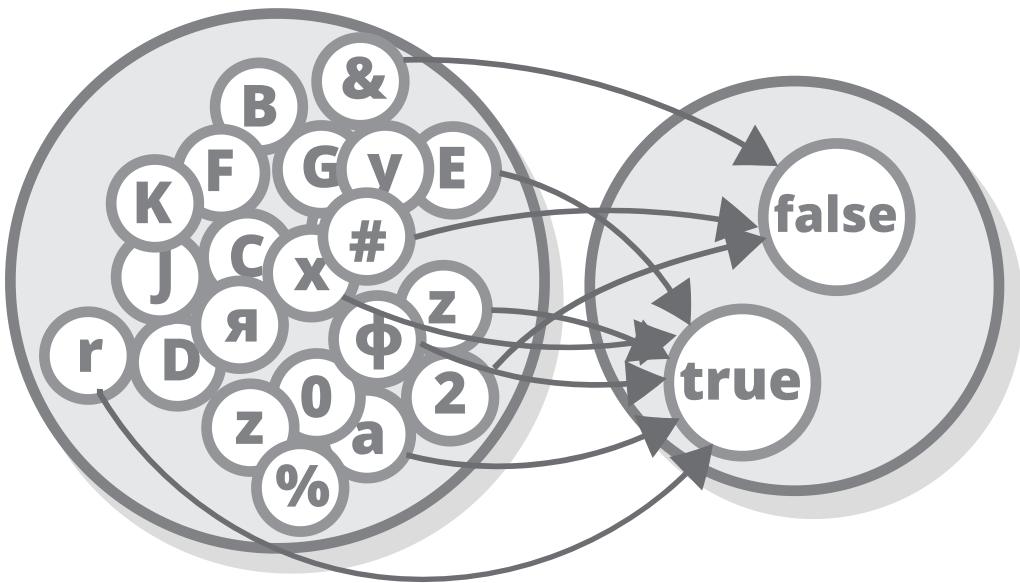
字符集合

大多数编程类型是复合类型——它们是这里列出的原始类型的组合。同样，我们稍后会讨论这些。

问题： 编程中的子集等价物是什么？

函数与方法/子程序 (Functions and methods/subroutines)

在编程中，一些函数（也称为方法、子程序等）有点类似于数学函数——它们有时接受一个给定类型的值（换句话说，属于给定集合的元素），并且总是返回一个属于另一个类型（或集合）的元素。例如，以下是一个接受 `Char` 类型的参数并返回 `Boolean` 的函数，用于指示该字符是否为字母。



从 Char 到 Boolean 的函数

然而，大多数编程语言中的函数也可能与数学函数完全不同——它们可以执行各种与返回值无关的操作。这些操作有时被称为副作用。

为什么编程中的函数不同呢？嗯，找到一种将有副作用的函数编码为数学上合理的方式并不容易，而且在当今大多数编程范式被创建时，人们面临的问题比函数不符合数学规范要大得多（例如，能够实际运行任何程序）。

如今，许多人认为数学函数过于限制且难以使用。他们可能是对的。但数学函数相对于非数学函数有一个很大的优势——它们的类型签名几乎告诉你关于函数的所有信息（这可能是大多数函数式语言强类型的原因除）。

纯函数式编程语言 (Purely-functional programming languages)

我们说过，虽然所有数学函数也是编程函数，但对于大多数编程语言，反之则不成立。然而，有些语言只允许数学函数，因此这种等价性成立。它们被称为纯函数式编程语言。

纯函数式编程语言的一个特点是，它们不支持执行诸如在屏幕上渲染内容、进行 I/O 等操作的函数（在此上下文中，这类操作被称为“副作用”）。

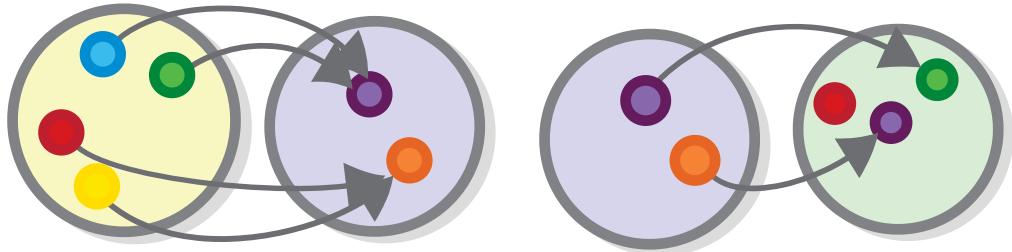
在纯函数式编程语言中，这类操作被委托给语言的运行时。与其编写直接执行副作用的函数，例如 `console.log('Hello')`，我们编写返回代表该副作用的类型的函数（例如，在 Haskell 中，副作用

由 `IO` 类型处理），然后运行时为我们执行这些函数。

然后，我们使用所谓的持续传递风格 (*continuation passing style*) 将所有这些函数链接成一个完整的程序。

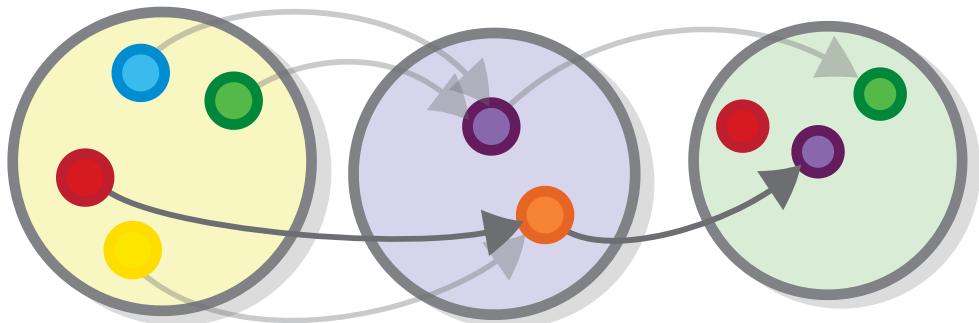
函数组合 (Functional Composition)

现在，我们即将进入函数主题的核心内容。那就是函数组合。假设我们有两个函数， $g : Y \rightarrow P$ 和 $f : P \rightarrow G$ ，它们的目标与源相同。



匹配的函数

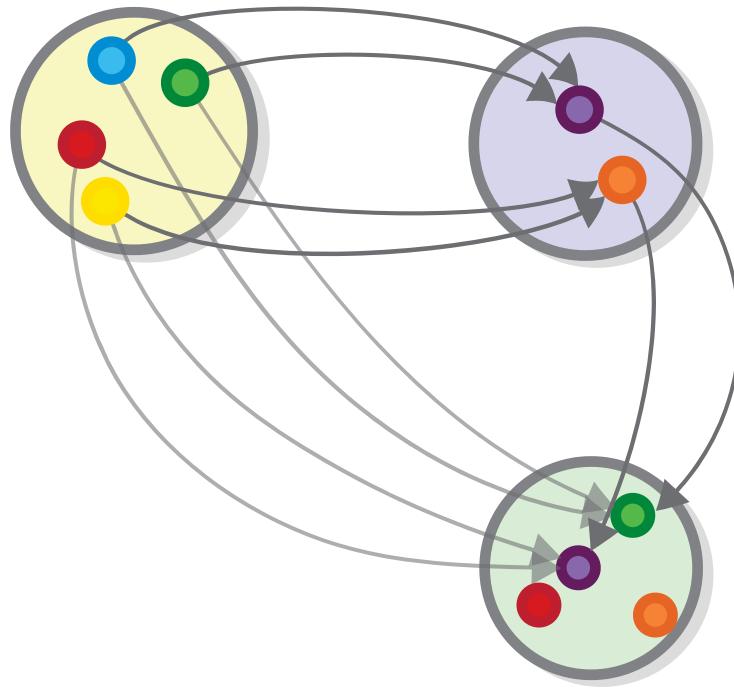
如果我们将第一个函数 g 应用于集合 Y 中的某个元素，我们将得到集合 P 中的一个元素。然后，如果我们将第二个函数 f 应用于那个元素，我们将得到集合 G 中的一个元素。



一个函数接一个函数应用

我们可以定义一个函数，它相当于执行上述操作的函数。这个函数是这样的，如果你遵循集合 Y 中的元素的箭头 h ，你将得到与遵循 g 和 f 箭头相同的集合 G 元素。

让我们称其为 $h : Y \rightarrow G$ 。我们可以说 h 是 g 和 f 的组合, 或 $h = f \square g$ (注意, 第一个函数在右边, 所以它类似于 $b = f(g(a))$)。



函数组合

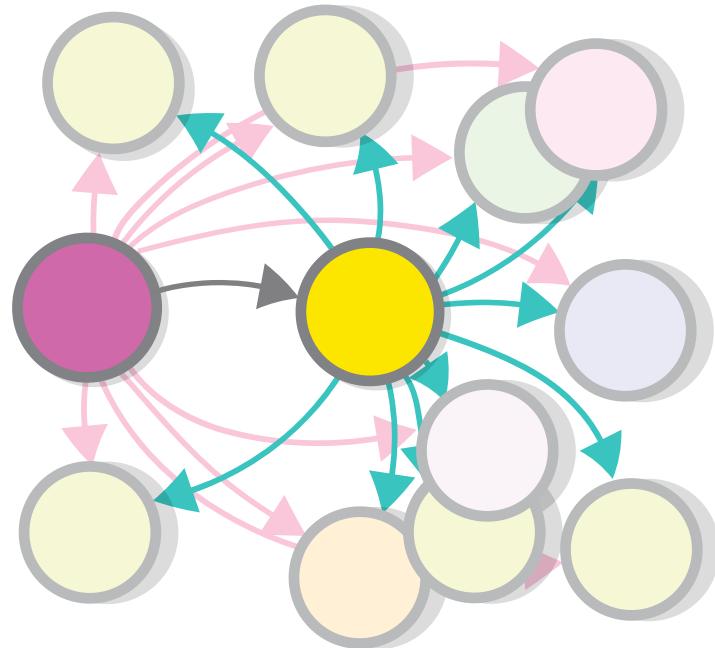
组合是所有范畴事物的本质。关键见解在于, 两个部分的总和并不比部分本身更复杂。

问题: 思考一下是什么使得函数可以进行组合, 例如, 它是否适用于其他类型的关系, 如多对多和一对多。

关系的组合 (Composition of relationships)

为了理解组合的强大功能, 考虑以下内容: 一个集合与另一个集合的连接意味着从第二个集合到任何其他集合的每个函数都可以转移到第一个集合的相应函数。

如果我们有一个函数 $g : P \rightarrow Y$, 那么对于从集合 Y 到任何其他集合的每个函数 f , 存在一个从集合 P 到相同集合的对应函数 $f \square g$ 。换句话说, 每当你定义一个从 Y 到其他集合的新函数时, 你就免费获得了一个从 P 到同一集合的函数。



函数组合连接

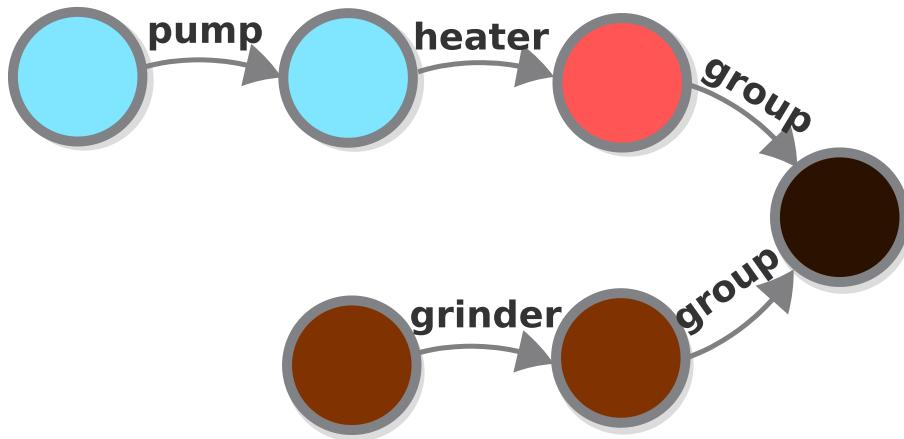
例如, 如果我们再次将人与其母亲之间的关系视为一个以世界上所有人为源集合, 所所有有孩子的人为目标集合的函数, 将此函数与其他类似函数组合将使我们获得一个人的母系亲属。

虽然你可能是第一次看到函数组合, 但它背后的直觉已经存在——我们都应该知道我们母亲的每个亲戚都是我们的亲戚——我们母亲的父亲是我们的祖父, 我们母亲的伴侣是我们的父亲, 等等。

工程中的组合 (Composition in engineering)

除了用于分析已存在的关系外, 组合原理还可以帮助你在构建表现出这种关系的对象(即工程)中发挥作用。

现代工程与古代手工业的主要区别之一是零件/模块/组件的概念——一种产品，它执行一个给定的功能，但并不是直接使用的，而是经过优化后与其他此类产品组合，形成“终端用户”产品。例如，浓缩咖啡机只是组件的组合，例如，泵、加热器、研磨组等，当以适当的方式组合时。



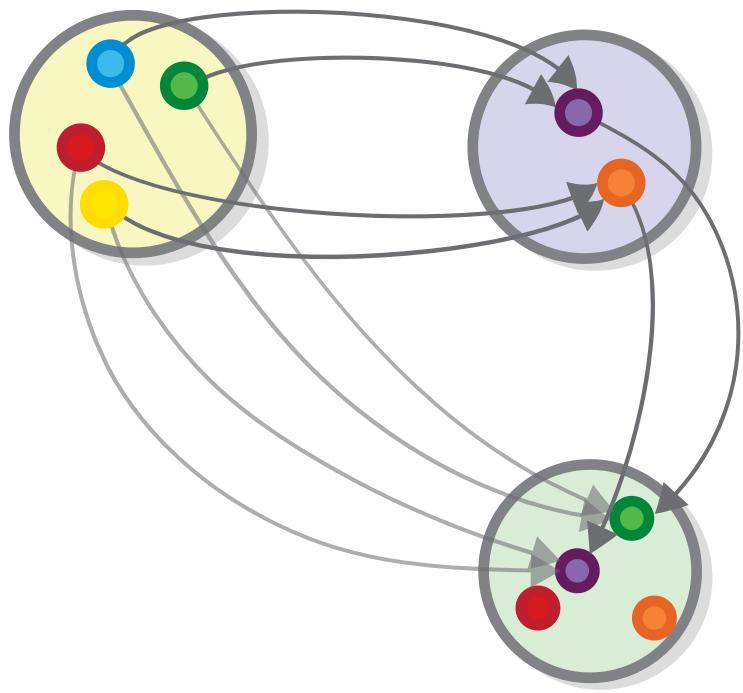
浓缩咖啡机

任务：思考这些函数的源和目标是什么。

顺便说一句，显示不显示集合元素的函数的“缩小”图称为外部图，与我们之前看到的内部图相对。

组合与外部图 (Composition and external diagrams)

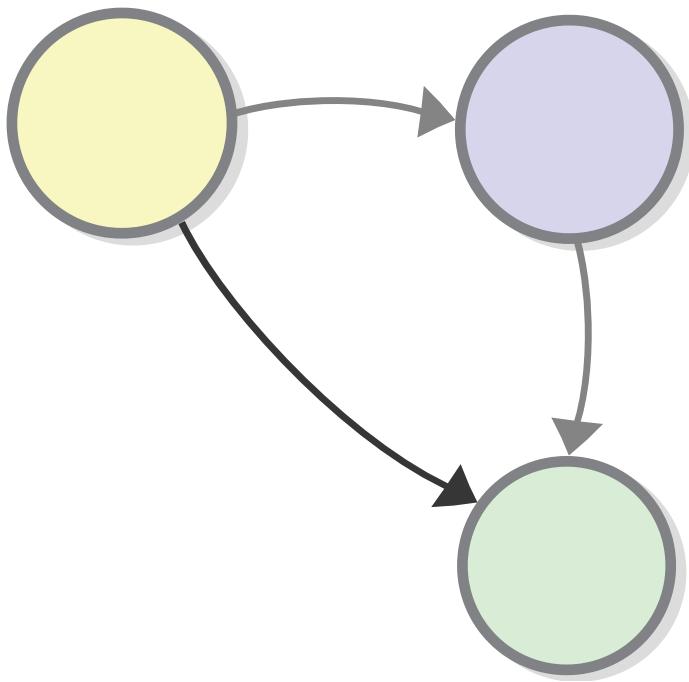
让我们看看演示函数组合的图，我们表明了组合的两个函数 ($f \square g$) 和新函数 (h) 的顺次应用是等价的。



函数组合

我们通过绘制一个内部图，并明确绘制函数源和目标的元素，以证明两条路径是等价的。

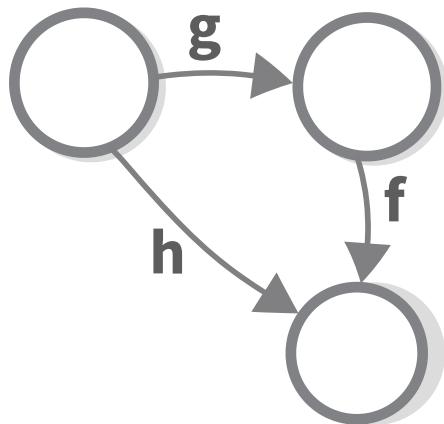
或者，我们可以简单地说箭头路径是等价的（从给定集合元素出发的所有箭头最终都会到达结果集合中的相同对应元素），并将等价关系绘制为一个外部图。



集合的函数组合

外部图是组合概念的更合适表示，因为它更通用。事实上，它如此通用，以至于它实际上可以作为函数组合的定义。

函数 f 和 g 的组合是一个第三个函数 h ，其定义为该图中的所有路径都是等价的。



函数组合 - 通用定义

如果你继续阅读本书，你会听到更多关于路径等价的图表（顺便说一下，它们被称为交换图 (*commuting diagrams*)）。

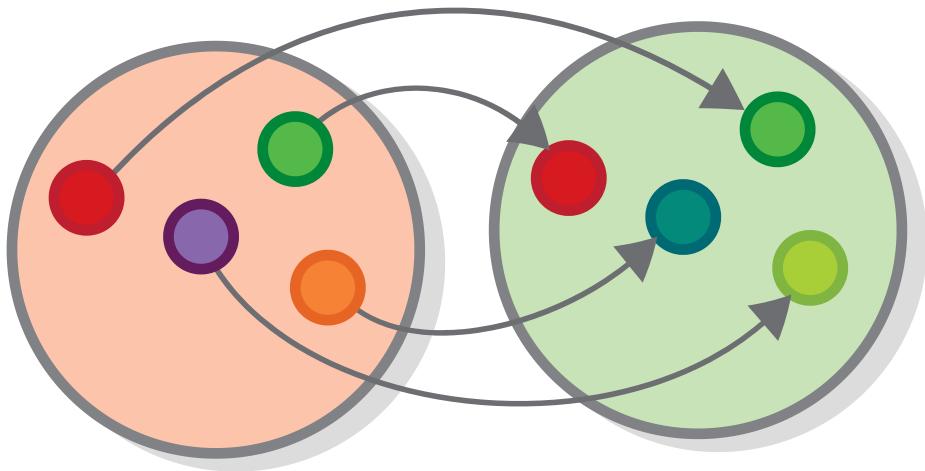
此时你可能会担心我忘记了我应该谈论范畴论，而只是展示了一些不相关的概念。我有时可能确实会这样做，但现在并不是这样——函数组合可以在不提及范畴论的情况下展示，但这并不能阻止它成为范畴论最重要的概念之一。

事实上，我们可以说（虽然这不是官方定义）范畴论是研究类似函数的事物（我们称它们为态射 (*morphisms*)）。它们有一个源和一个目标，以关联的方式组合，并且可以通过外部图表示。

还有另一种定义范畴论的方法，即用同构 (isomorphism) 的概念代替相等的概念。我们尚未讨论同构，但这将是我们本章剩余时间的内容。

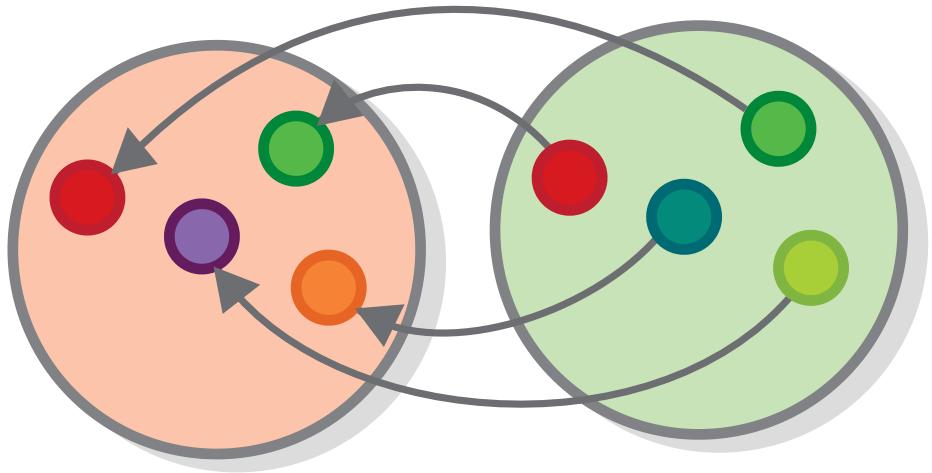
同构 (Isomorphism)

要解释什么是同构, 我们回到函数可以表示的关系类型的例子, 以及最基础的关系——一对一关系。我们知道, 所有函数都有一个源集合中的元素指向目标集合中的一个元素。但对于一对一函数, 反过来也成立——目标集合中的一个元素指向源集合中的一个元素。



相反的颜色

如果我们有一个连接大小相同集合的一对一函数(如这里的情况), 那么该函数具有以下性质: 目标集合中的所有元素都有一个箭头指向它们。在这种情况下, 该函数是可逆的。也就是说, 如果你翻转该函数的箭头及其源和目标, 你会得到另一个有效的函数。



相反的颜色

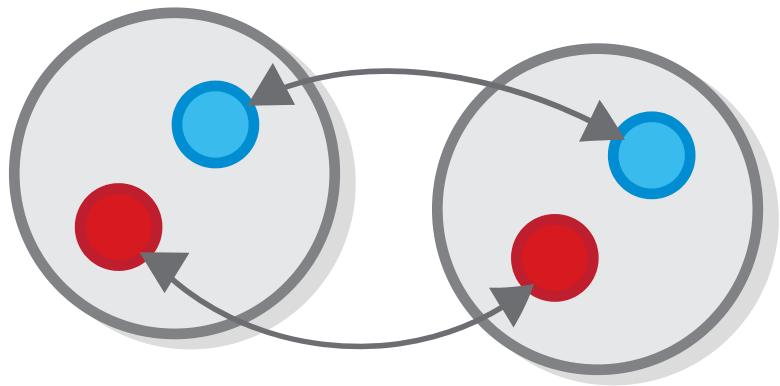
可逆函数被称为同构。当两个集合之间存在可逆函数时，我们说这两个集合是同构的。例如，因为我们有一个将摄氏温度转换为华氏温度并反之亦然的可逆函数，我们可以说摄氏温度和华氏温度是同构的。

同构在希腊语中的意思是“相同的形式”（虽然实际上两组同构集合之间唯一不同的就是它们的形式）。

更正式地说，两个集合 R 和 G 是同构的（或 $R \cong G$ ），如果存在函数 $f: G \rightarrow R$ 及其逆函数 $g: R \rightarrow G$ ，使得 $f \square g = ID_R$ 和 $g \square f = ID_G$ （注意恒等函数在这里派上了用场）。

同构与恒等 (Isomorphism and identity)

如果你仔细观察，你会发现恒等函数也是可逆的（它的逆函数是它自己），因此每个集合都是这样同构于自身的。

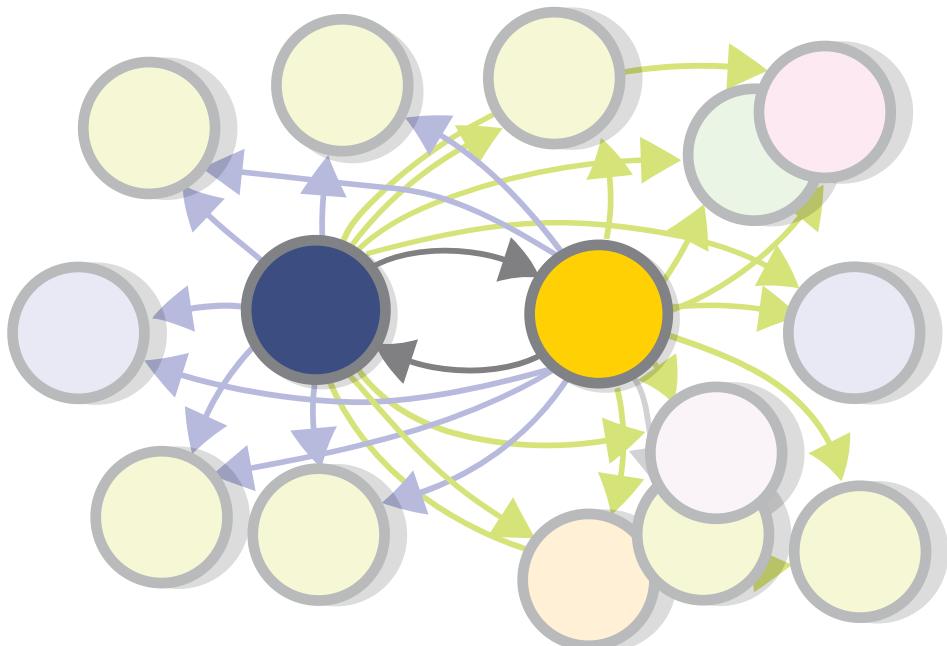


恒等函数

因此, 同构的概念包含了相等的概念——所有相等的事物也是同构的。

同构与组合 (Isomorphism and composition)

同构的一个有趣的事是, 如果我们有将集合 A 的成员转换为集合 B 成员的函数, 反之亦然, 那么由于函数组合的存在, 我们知道从/到 A 的任何函数都有一个相应的从/到 B 的函数。



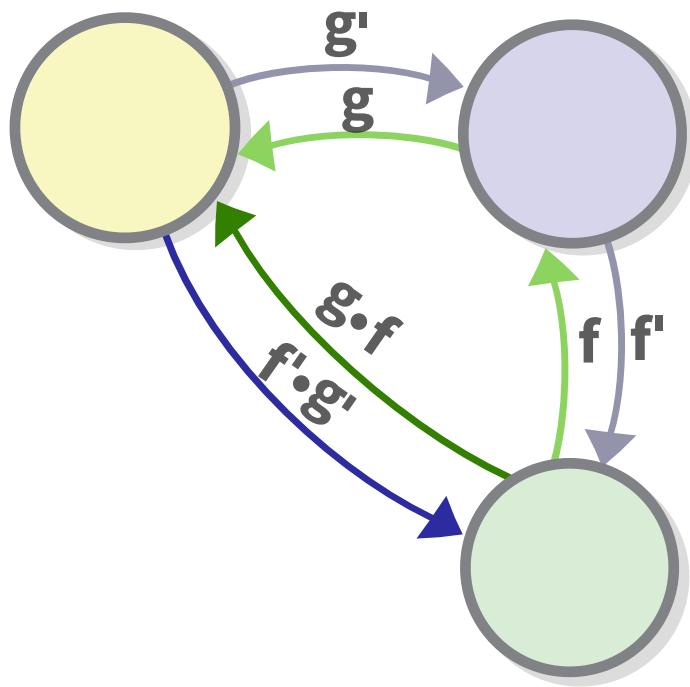
同构的架构

例如,如果你有一个从所有已婚人的集合到相同集合的“伴侣是”函数,那么该函数是可逆的。这并不是说你和你的伴侣是同一个人,而是说关于你或你与其他人或对象的每一个陈述也是他们与该人或对象的关系,反之亦然。

组合同构 (Composing isomorphisms)

关于同构的另一个有趣事实是,如果我们有两个共享集合的同构,那么我们可以通过组合(同构)获得其他两个集合之间的第三个同构。

将两个同构组合成另一个同构是通过将同构在两个方向上的两个函数组合来实现的。



组合同构

非正式地,我们可以看到这两个态射确实是彼此的逆函数,因此形成同构。如果我们想要正式地证明这一事实,我们会做类似于以下的事情:

鉴于如果两个函数是同构的，那么它们的组合等于恒等函数，证明函数 $g \square f$ 和 $f' \square g'$ 是同构的等同于证明它们的组合等于恒等。

$$g \square f \square f' \square g' = id$$

但我们已经知道 f 和 f' 是同构的，因此 $f \square f' = id$ ，所以上述公式等同于（你可以参考图表看看这意味着什么）：

$$g \square id \square g' = id$$

我们知道任何与 id 组合的东西等于它自身，因此它等同于：

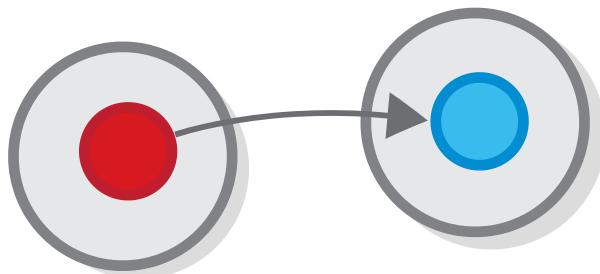
$$g \square g' = id$$

这是真的，因为 g 和 g' 是同构的，并且组合的同构函数等于恒等。

顺便说一下，还有另一种获得同构的方法——通过一个方向组合两个态射以获得第三个函数，然后再取它的逆函数。但要做到这一点，我们必须证明从组合两个双射函数得到的函数也是双射的。

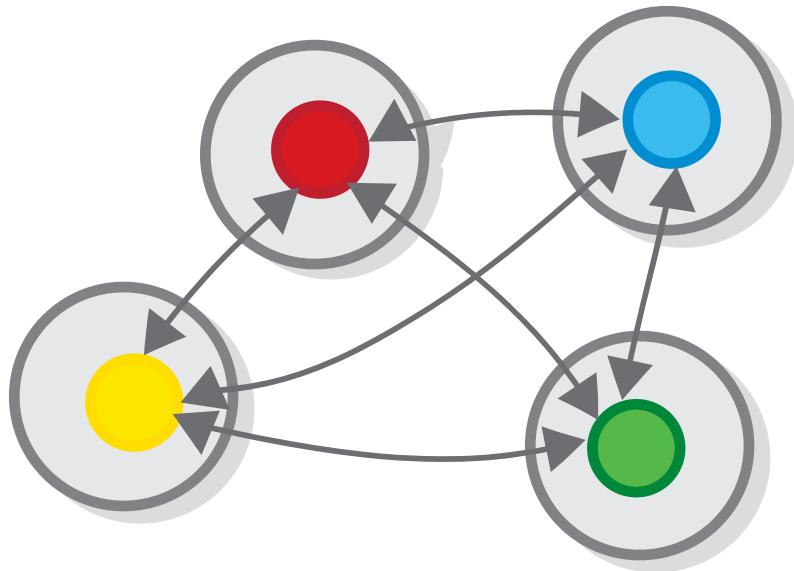
单例集合之间的同构 (Isomorphisms Between Singleton Sets)

在任何两个单例集合之间，我们可以定义唯一可能的函数。



单例之间唯一可能的函数

该函数是可逆的，这意味着所有单例集合是同构的，此外（这很重要）它们以唯一的方式同构。



同构的单例

按照最后一段的逻辑，关于独一无二的事物的每一个陈述都可以转移为关于另一个独一无二的事物的陈述。

问题：尝试提出一个好例子，展示如何证明单例集合之间同构的陈述（显然我想不出来）。考虑到所有人和物体共享同一个宇宙。

等价关系与同构 (Equivalence relations and isomorphisms)

我们说过，同构集合不一定是同一个集合（虽然反过来成立）。然而，很难摆脱这样的想法，即同构意味着它们在某种程度上是相等的或等价的。例如，所有通过同构母子关系连接的人共享一些相同的基因。

在计算机科学中，如果我们有将对象 A 转换为对象 B 并反之亦然的函数（例如数据结构及其 id 之间的函数），我们也可以基本上将 A 和 B 视为同一事物的两种格式，因为拥有其中一个意味着我们可以轻松获得另一个。

等价关系 (Equivalence relations)

两个事物等价意味着什么？这个问题听起来很哲学化，但实际上有一个正式的方式来回答它，即存在一个捕捉相等概念的优雅数学概念——等价关系的概念。

那么什么是等价关系呢？我们已经知道关系是什么——它是两个集合之间的连接（一个例子是函数）。但什么时候关系是等价关系呢？根据定义，它符合三个直观的关于相等的法律。让我们回顾一下它们。

自反性 (Reflexivity)

定义等价的第一个思想是每件事物都与其自身等价。

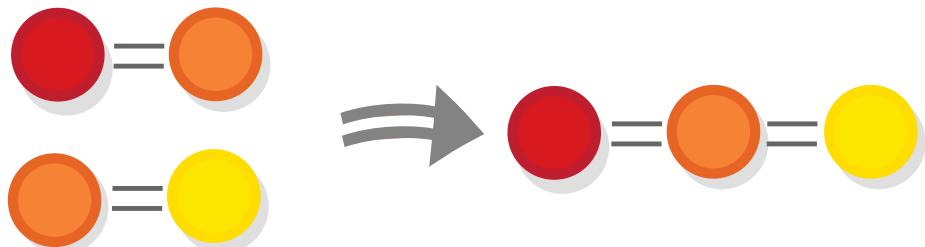


自反性

这个简单的原则可以转化为同样简单的**自反性定律**:对于所有集合 A ,
 $A = A^{\circ}$ 。

传递性 (Transitivity)

根据基督教的圣三一神学,耶稣的父亲是上帝,耶稣是上帝,圣灵也是上帝,然而,父亲与耶稣不是同一个人(耶稣也不是圣灵)。如果你觉得这很奇怪,那是因为它违反了等价关系的第二条定律——传递性。传递性是指等于第三件事物的两件事物必须彼此相等。



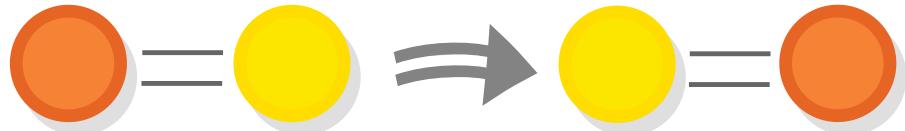
传递性

数学上,对于所有集合 $A \cdot B$ 和 C ,如果 $A = B$ 且 $B = C$,那么 $A = C$ 。

注意,我们不需要定义涉及三个以上集合的类似情况,因为它们可以通过多次应用该定律来解决。

对称性 (Symmetry)

如果一件事物等于另一件事物，反过来也成立，即另一件事物也等于第一件事物。这一思想被称为对称性。对称性可能是等价关系的最典型特性，而其他几乎没有关系具有这一特性。



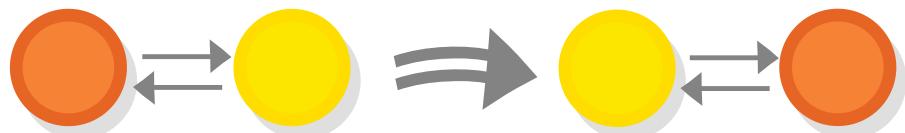
对称性

用数学术语来说：如果 $A = B$ 那么 $B = A$ 。

同构作为等价关系 (Isomorphisms as equivalence relations)

同构确实是等价关系。而且“顺便说一下”，我们已经有了证明它所需的所有信息（就像詹姆斯·邦德总是碰巧有完成任务所需的所有装备一样）。

我们说过等价关系的最典型特性是其对称性。而由于同构的最典型特性，即它们是可逆的，这一特性也在同构中得到了满足。



同构的对称性

任务：一条定律已解决，还有两条：请查看前一节内容，验证同构是否也符合其他等价关系定律。

使用同构定义等价关系的做法在范畴论中非常显著，在范畴论中同构用符号 \cong 表示，它几乎与 $=$ 相同（也类似于用两条相反的箭头连接一个集合到另一个集合）。

插曲：同构与数字 (Interlude — numbers as isomorphisms)

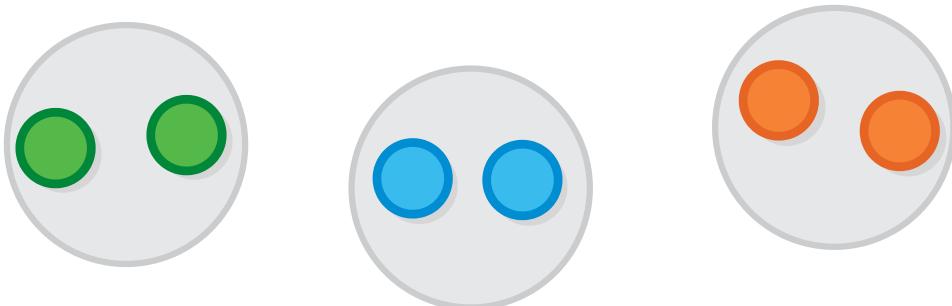
许多人会说，数字的概念是数学中最基本的概念。但实际上他们错了——集合和同构更加基本！或至少，数字可以通过集合和同构来定义。

要理解这一点，让我们思考一下你如何教一个人数字的概念（特别是，这里我们将集中讨论自然数，即计数用的数字）。你可以通过展示一些给定数量的物体来开始教学，例如，如果你想展示数字 2，你可能会拿出两支铅笔、两个苹果或其他两个东西。



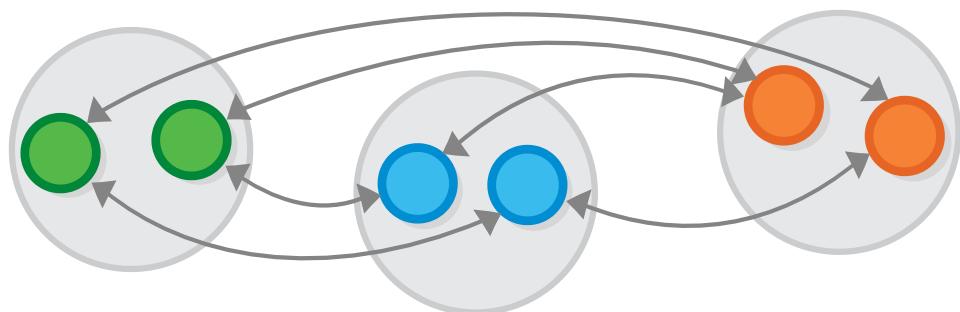
两个球

在你这样做时，重要的是要强调你并不是指左边的物体，或者仅仅指右边的物体，而是要考虑这两样东西在一起，换句话说，就是将它们视为一个整体。因此，如果你正在向一个已经了解集合概念的人解释这个问题，这个知识点可能派得上用场。此外，作为好的教师，我们可能会提供一些包含两个事物的集合的其他例子。



两球的集合

这是一个很好的起点，但学习者可能仍然盯着这些物体而不是关注结构——他们可能会问，其他这个或那个集合也是 2 吗？此时，如果你解释的对象碰巧知道同构（假设他们曾经在洞穴中与这本书相伴），你可以很容易地进行最终的定义，告诉他们数字 2 是由这些集合和所有与它们同构的其他集合代表的，或者按照正式定义，它是具有两个元素的集合的等价类。



两个球的同构集合

此时，我们不再需要添加更多的例子。事实上，因为我们考虑了所有其他集合，我们可以说这不仅仅是一堆例子，而是数字 2 的一个恰当的定义。我们可以将这种方法扩展到包括所有其他数字。事实上，第一种自然数的定义（由戈特洛布·弗雷格 (Gottlob Frege) 于1884年提出）大致基于这个想法。

在结束本章之前，我们必须指出一个元注：根据我们提出的数字定义，数字并不是一个对象，而是一个包含无数相互关联对象的系统。这对你来说可能有些奇怪，但它实际上是范畴论中建模方式的一个典型特征。

附录:软件开发中的组合案例

(Addendum: The case of composition in software development)

“不结构化的单片设计不是一个好主意,除了可能对于像烤面包机这样的小型操作系统,甚至在这种情况下也是有争议的。”—— 安德鲁·S·塔宁鲍姆 (Andrew S. Tanenbaum)

软件开发是一门特殊的学科——理论上,它应该是某种工程,但实际上执行时,它有时更接近于手工艺,没有充分利用组合原则。

想象一个人(例如我自己)在修理某个工程问题,例如试图修理一台机器或修改它以适应新的用途。如果这台机器是机械的或电气的,那么这个人将不得不利用现有的组件,因为他们不太可能自己制造新组件(或至少他们会避免这么做)。这种限制迫使组件制造商创建通用且协作良好的组件,就像纯函数组合良好一样。这反过来又使工程师能够更轻松地制造出更好的机器,而不必自己完成所有工作。

但如果这台机器是基于软件的,情况就不同了——由于新软件组件可以很容易地被创建,我们的设计可以模糊组件之间的界限,甚至完全消除组件的概念,将整个程序变成一个巨大的组件(即单片设计)。更糟糕的是,当没有现成的组件可用时,这种方法实际上比基于组件的设计更容易使用,因此许多人选择了它。

这很糟糕,因为单片设计的好处主要是短期的——没有拆分成组件的程序更难以理解、更难修改(例如,你不能用一个新的组件替换一个故障组件),总体上比基于组件的程序更原始。出于这些原因,我认为目前程序员没有充分利用函数组合的原则。实际上,我对此感到如此不满,以至于决定写一本关于应用范畴论的书,帮助人们更好地理

解组合的原则——它叫做《范畴论图解》(Category Theory Illustrated)
(哦, 等等, 我现在正在写这本书, 对吧?)

从集合到范畴 (From Sets to Categories)

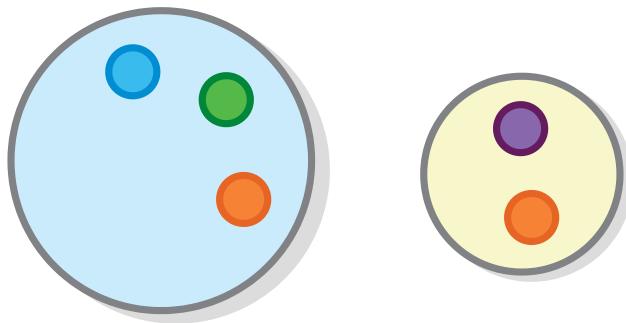
在本章中, 我们将看到一些更多的集合论构造, 同时也将引入它们的范畴论 (category-theoretic) 对应物, 以便轻松引入范畴 (category) 概念本身。

当我们完成这些内容后, 我们将尝试(并几乎成功地)从头开始定义范畴, 而不依赖于集合论。

积 (Products)

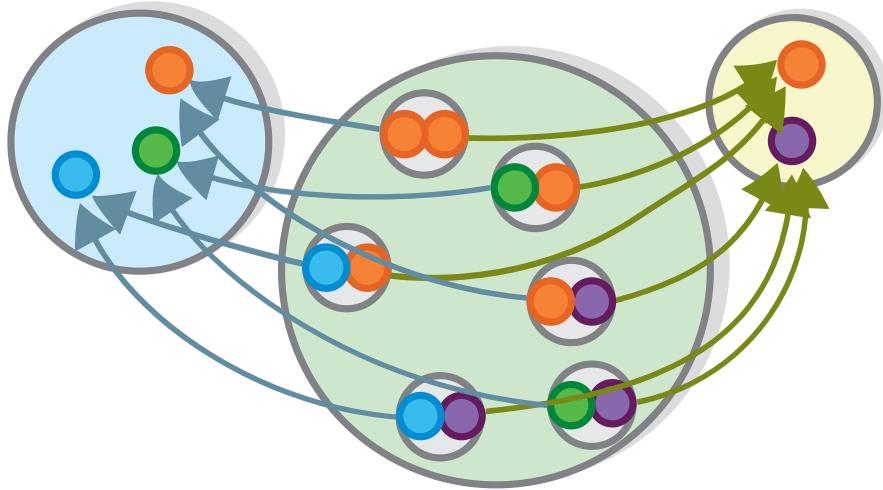
在上一章中,有几个地方我们需要构造一个集合,其元素是一些其他集合元素的复合。例如,当我们讨论数学函数时,由于只能构建接受一个参数的函数,因此我们无法定义加法 (+) 和减法 (-)。然后,当我们介绍编程语言中的基本类型(如 `Char` 和 `Number`)时,我们提到,实际上我们使用的大多数类型都是复合类型。那么我们如何构建这些复合类型呢?

最简单的复合类型是集合 B ,其中包含 b 的元素,集合 Y ,其中包含 y 的元素。这就是 B 和 Y 的笛卡尔积(Cartesian product),即包含一个来自集合 Y 的元素和一个来自集合 B 的元素的有序对。形式化地表示: $Y \times B = \{(y, b)\}$,其中 $y \in Y, b \in B$ (\in 表示“是...的元素”。



积的部分

它表示为 $B \times Y$,并配备有两个函数,分别用于从每个 (b, y) 中检索 b 和 y 。



积

问题：为什么它被称为“积”呢？提示：它有多少个元素？

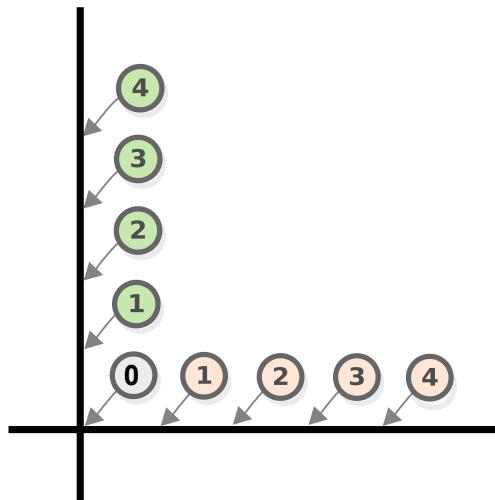
```
{% if site.distribution == 'print'%}
```

插曲——坐标系 (Interlude — coordinate systems)

笛卡尔积 (Cartesian product) 的概念首先由数学家和哲学家勒内·笛卡尔 (René Descartes) 定义，作为笛卡尔坐标系 (Cartesian coordinate system) 的基础，这也是为什么这两个概念都以他的名字命名（尽管表面上看不出来，因为它们使用的是他名字的拉丁化形式）。

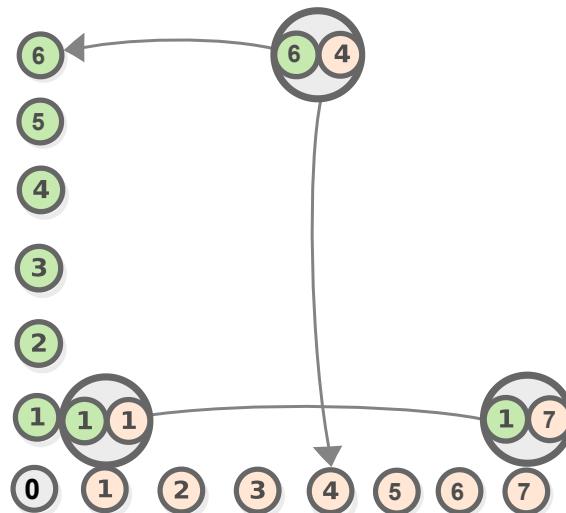
大多数人都知道笛卡尔坐标系是如何工作的，但很少有人思考如何用集合和函数来定义它。

笛卡尔坐标系由两条垂直的直线组成，这些直线位于欧几里得平面 (Euclidean plane) 上，并通过类似函数的映射，将平面上的任意一点与一个数值相关联，该数值表示该点与两条直线交点的距离（该交点被映射到数值 0）。



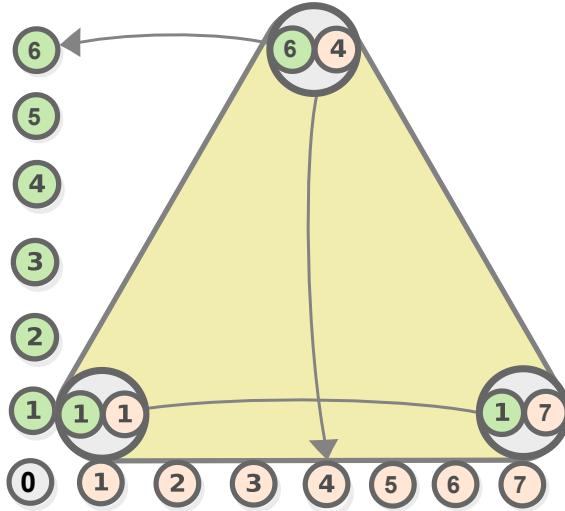
笛卡尔坐标

利用这一构造(以及笛卡尔积的概念), 我们不仅可以描述这些直线上的点, 还可以描述欧几里得平面上的任意一点。我们通过测量该点与两条直线的距离来做到这一点。



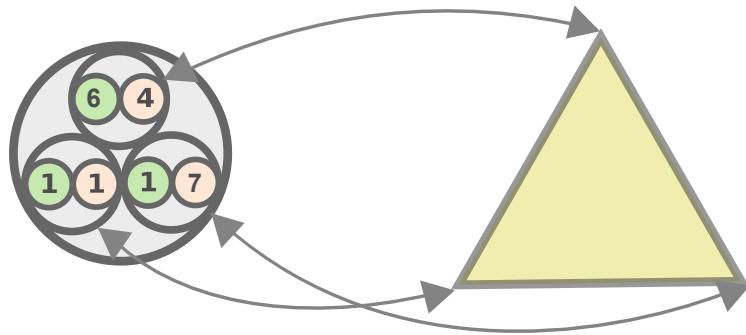
笛卡尔坐标

由于点是欧几里得几何的主要原始对象, 坐标系还允许我们描述所有类型的几何图形, 例如这个三角形(它是通过多个积的形式来描述的)。



笛卡尔坐标系的三角形

因此,我们可以说,笛卡尔坐标系是某种类似函数的映射,它将所有种类的(积的)数的积集合与这些数对应的几何图形联系起来,利用这些数,我们可以推导出图形的一些性质(例如,通过下图中的积,我们可以计算出这个三角形的底边长为6个单位,高为5个单位)。



笛卡尔坐标系同构

更有趣的是,这种映射是一对一的,这使得两个领域同构(isomorphic)(传统上,我们说该点被坐标完全描述,这意味着同样的事情)。

到此为止,我们通过集合来表示笛卡尔坐标系的工作基本完成,但我们还没有描述将点与数值连接起来的这些类似函数的东西——它们直观地理解为函数,并且表现出所有相关的性质(例如,多对一映射,或者在这种情况下是一对一映射)。然而,我们只讨论了集合之间的函数映射。即使我们可以将坐标系视为一个集合(由点和图形组成),几何图形显

然不是集合,因为它包含了很多附加内容(或者用范畴论的说法,附加的结构)。因此,正式定义这种映射,需要我们同时形式化几何和代数,并使它们彼此兼容。这也是范畴论的一些雄心壮志,稍后我们将在本书中尝试解决这一问题。

在我们继续讨论之前,让我们看看积的一些其他有趣用途。

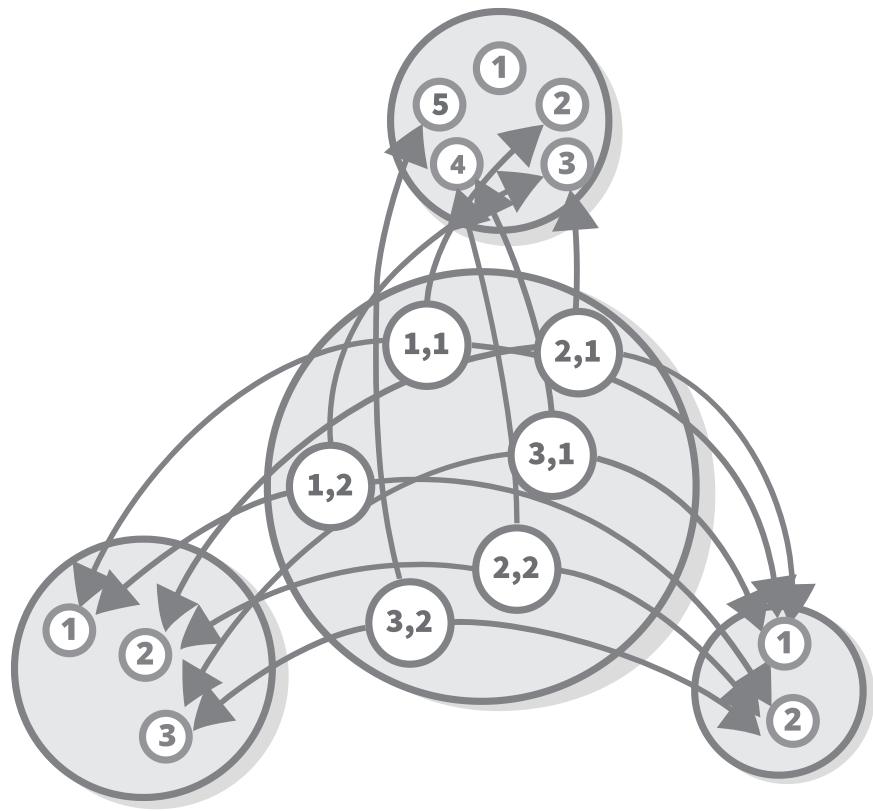
{%endif%}

作为对象的积 (Products as Objects)

在上一章中,我们建立了编程语言概念和集合论的对应关系——集合类似于类型,函数类似于方法/子例程。这个图景通过积得以完善,积就像简化版的类(class)(也称为记录(record)或结构体(struct))——形成积的集合对应于类的属性(properties)(也称为成员(members)),用于访问这些属性的函数就像程序员所说的*getter*方法。例如,面向对象编程中著名的 Person 类,具有 name 和 age 字段,这实际上不过是字符串集合和数字集合的积。具有多个值的对象可以通过积的复合来表达,而这些复合本身也是积。

使用积定义数值运算 (Using Products to Define Numeric Operations)

积还可以用于表达接受多个参数的函数(事实上,这也是多参数函数在具有元组(tuple)的语言中实现的方式,如 ML 家族的语言)。例如,“加法”是从两个数字的积集合到数字集合的函数,因此, $+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ 。



加法函数

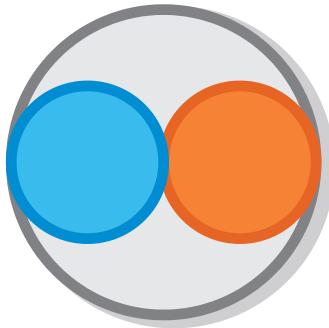
换句话说，积是非常重要的概念，如果你想表示任何类型的结构，它都是必不可少的。

通过集合定义积 (Defining products in terms of sets)

如我们所说，积是有序对的集合（形式上讲 $A \times B \neq B \times A$ ）。因此，为了定义积，我们必须定义有序对的概念。那么我们如何做到这一点呢？请注意，有序对的元素不仅仅是一个包含两个元素的集合——那只会是一个对，而不是一个有序对。

相反，有序对是一种结构，它包含两个对象以及关于哪一个是第一个、哪一个是第二个的信息（在编程中，我们可以为对象的每个成员指定名称，这与对的排序功能相同）。

有序部分很重要，因为虽然某些数学运算（如加法）不关心顺序，但其他运算（如减法）则非常依赖顺序（在编程中，当我们操作对象时，我们显然希望访问特定属性，而不是随机的属性）。



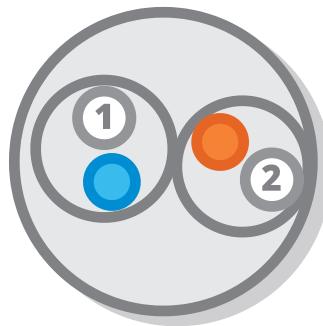
一个对

这是否意味着我们必须将有序对定义为类似集合的“原始”类型才能使用它们？这有可能，但还有另一种方法：如果我们能够使用仅由集合定义的构造来表示有序对的同构物，我们就可以使用该构造代替有序对。数学家已经提出了多种聪明的方法来做到这一点。以下是第一个方法，由诺伯特·维纳 (Norbert Wiener) 于1914年提出。请注意空集唯一性的巧妙运用。



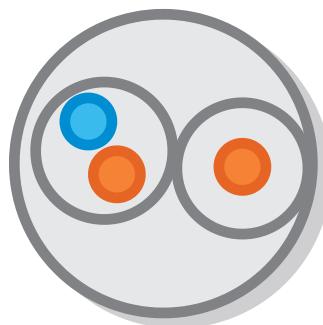
由集合表示的对

下一种方法同样在1914年由费利克斯·豪斯多夫 (Felix Hausdorff) 提出。要使用这种方法，我们首先必须定义 1 和 2。



由集合表示的对

1921年，卡齐米日·库拉托夫斯基 (Kazimierz Kuratowski) 提出的这种方法只使用了对的组成部分。

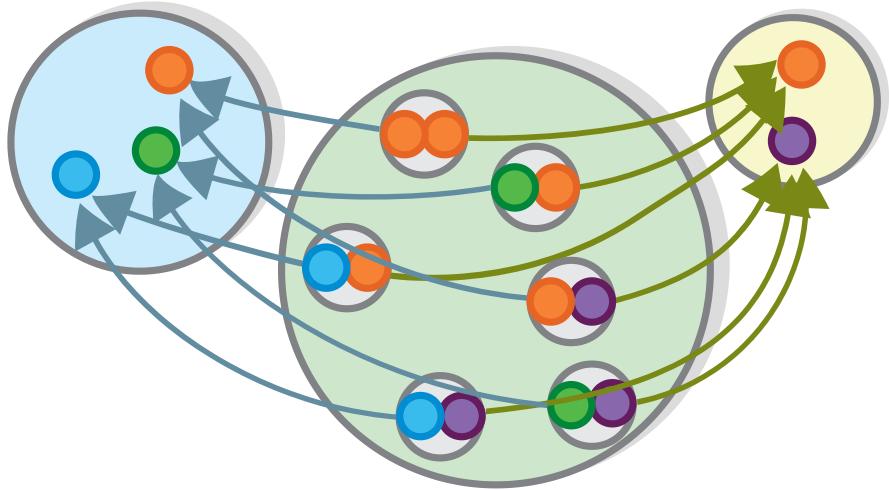


由集合表示的对

通过函数定义积 (Defining products in terms of functions)

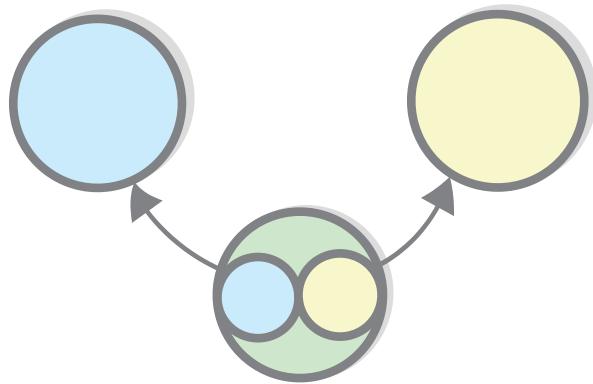
前面的积定义通过放大积的各个元素来查看它们的组成部分。我们可以将其视为定义的低层次方法。这次我们将尝试做相反的事情——尽可能地忽视我们的集合的内容，即我们不会放大，而是缩小，尝试避开我们在前一部分遇到的困难，并提供基于函数和外部图表的积定义。

我们如何通过外部图表定义积呢？即给定两个集合，如何确定它们的积集合？为此，我们首先必须考虑积有哪些函数，我们有两个函数——分别用于检索对的两个元素（所谓的“getter”）。



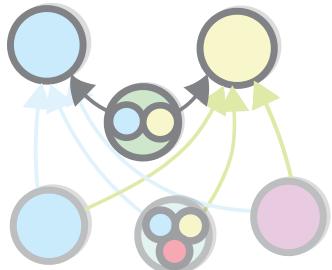
积

形式上,如果我们有一个集合 G ,它是集合 Y 和 B 的积,那么我们还应该有函数返回积的元素,因此 $G \rightarrow Y$ 和 $G \rightarrow B$ 。现在让我们切换到外部视角。



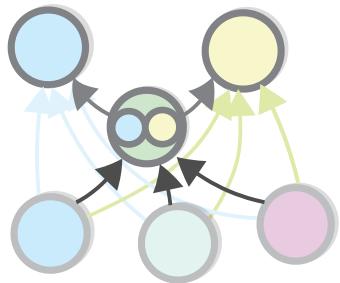
积, 外部图表

这个图表已经提供了一个定义,但还不是一个完整的定义,因为 Y 和 B 的积并不是唯一可以定义这些函数的集合。例如, $Y \times B \times R$ 的三元组集合,对于任何元素 R 也符合条件。如果存在从 G 到 B 的函数,那么集合 G 本身也满足我们的积条件,因为它与 B 和它自身相连。还有许多其他对象也符合条件。



积，外部图表

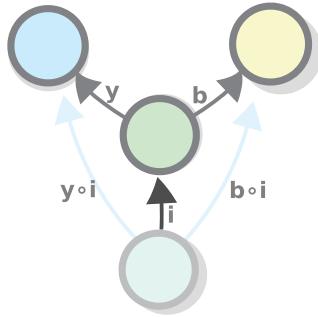
然而，三元组集合 $Y \times B \times R$ 仅仅因为它可以被转换为 $Y \times B$ ，因此与 Y 和 B 相连：箭头 $Y \times B \times R \rightarrow B$ 只是箭头 $Y \times B \times R \rightarrow Y \times B$ 和箭头 $Y \times B \rightarrow B$ 的复合。相同的推理适用于所有其他对象。



积，外部图表

(直观上，所有这些对象都比对的对象更复杂，你总是可以有一个函数将更复杂的结构转换为更简单的结构（当我们讨论将子集转换为它们的超集的函数时，我们已经看到了一个例子）。)

更正式地，如果我们假设有一个集合 I 可以作为集合 B 和 Y 的假积 (impostor product)，即 I 是这样一个集合，存在两个函数，我们称之为 $ib : I \rightarrow B$ 和 $iy : I \rightarrow Y$ ，它们允许我们从中得出 B 和 Y 的元素，那么还必须存在一个类型签名为 $I \rightarrow B \times Y$ 的唯一函数，该函数将假积转换为真积， ib 和 iy 只是将该函数与通常的“getter”函数复合的结果，该 getter 函数将从对中返回元素（即无论我们选择哪个对象作为 I ，此图表都会交换）。



积的普遍性质

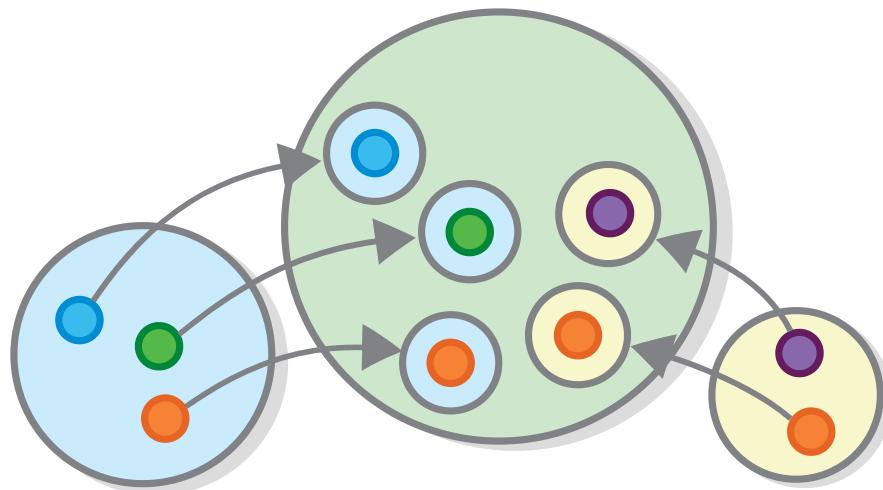
在范畴论中,这种给定对象可能具有的性质(参与某种结构,使得所有类似的对象都可以转换为/从它转换而来)称为**普遍性质**(universal property)。我们不会对此进行更详细的讨论,因为现在讨论这个问题还为时过早(毕竟我们还没有真正解释范畴论是什么)。

我们需要指出的一件事是,这个定义(顺便说一下,前面的所有定义也是如此)并没有排除与积同构的集合——当我们使用普遍性质表示事物时,同构与等同是一回事。在编程中,我们必须采用相同的观点,尤其是在我们处理更高级别的事物时——可能存在许多不同的对的实现(例如,由不同库提供的实现),但只要它们以相同的方式工作(即我们可以将一个转换为另一个,反之亦然),它们对我们来说都是相同的。

和积 (Sums)

我们现在将研究一个与积 (product) 非常相似但同时也非常不同的构造。它之所以相似，是因为它也是两个集合之间的一种关系，允许你将它们合并为一个集合，而不抹去它们的结构。但不同之处在于，它编码了一种完全不同的关系——积编码的是两个集合之间的与关系，而和编码的是或关系。

两个集合 B 和 Y 的和，表示为 $B + Y$ ，是一个包含第一个集合中的所有元素与第二个集合中的所有元素的集合。



和或上积

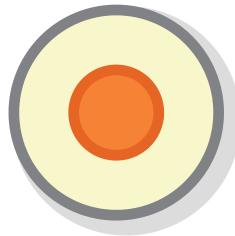
我们可以立即看到它与或逻辑结构的联系：例如，因为父母是孩子的母亲或父亲，所有父母的集合就是母亲集合和父亲集合的和，或者 $P = M + F$ 。

用集合表示和 (Defining Sums in Terms of Sets)

与积一样，使用集合表示和并不是那么简单。例如，当某个对象是两个集合的元素时，它会在和中出现两次，这是不允许的，因为一个集合不能包

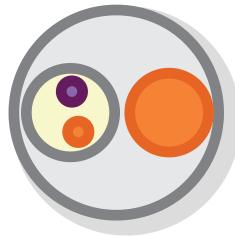
含相同的元素两次。

与积一样,解决方案是增加一些额外的结构。



和的成员

与积一样,有一种低层次的方法可以使用仅有的集合来表示和。巧合的是,我们可以使用对(pair)。



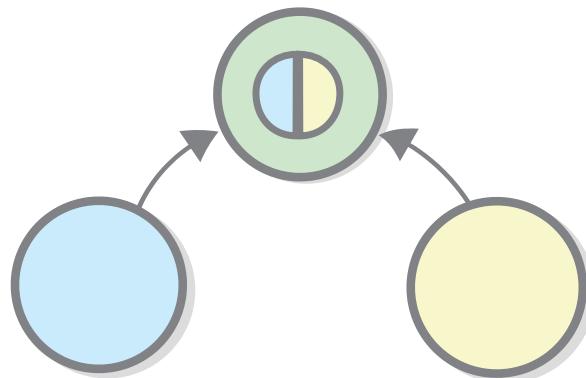
和的成员,分析

用函数表示和 (Defining Sums in Terms of Functions)

正如你可能已经怀疑到的,有趣的部分是使用函数表示两个集合的和。为此,我们必须回到定义的概念部分。我们说过,和表示的是两个事物之间的或关系。

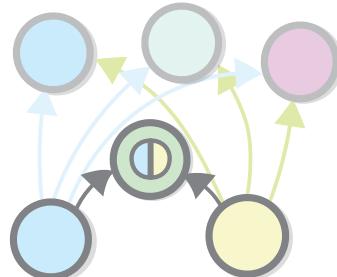
每个或关系的一个性质是:如果某物是 A ,那么它也是 $A \vee B$, B 也是如此(顺便说一句,符号 \vee 表示“或”)。例如,如果我的头发是棕色,那么我的头发也是要么金色,要么棕色。这就是或的意思,对吧?这个性质可以表示为一个函数,实际上有两个函数——每个参与和关系的集合都有

一个函数（例如，如果父母是母亲或父亲，那么肯定存在 $mothers \rightarrow parents$ 和 $fathers \rightarrow parents$ 的函数）。



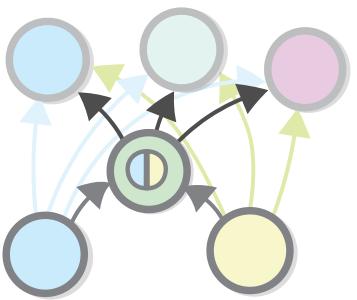
和积, 外部图表

你可能已经注意到，这个定义与前一部分中积的定义非常相似。而且相似之处不止于此。与积一样，我们有些集合可以被视为假和 (impostor sums)——这些函数存在，但它们还包含额外的信息。



和积, 外部图表

所有这些集合表达的关系比简单的和要模糊得多，因此，给定这样一个集合（如前面所说的“假集合”），将会存在一个唯一的函数区分它与真实的和。唯一的区别是，与积不同，这次这个函数是从和到假集合的。

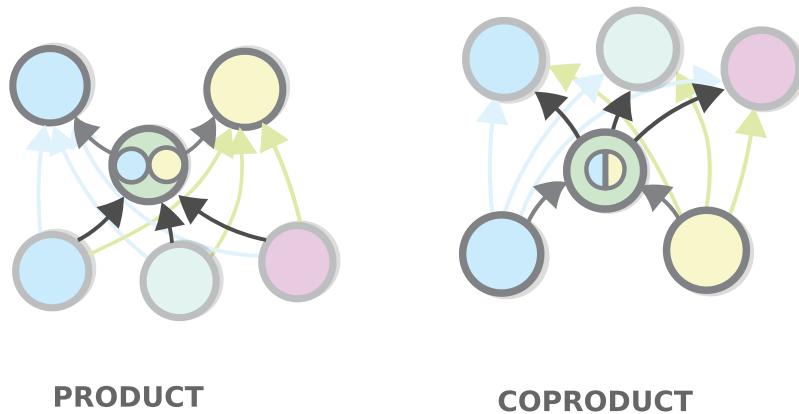


和积, 外部图表

插曲：范畴对偶 (Interlude: Categorical Duality)

从内部图表来看，和 (sum) 和积 (product) 可能看起来已经有些相似了，但一旦我们切换到外部视角并绘制这两个概念的外部图表，这种相似性就会立即变得清晰。

我使用了复数形式的“图表”，但实际上这两个概念是通过同一个图表捕捉到的，唯一的区别是它们的箭头方向相反——多对一的关系变成了一对多，反之亦然。



和积与积的对偶性

定义这两个构造的普遍性质 (universal properties) 也是相同的——如果我们有一个和 $Y + B$, 对于每个假和 (impostor sum), 如 $Y + B + R$, 存在一个平凡函数 $Y + B \rightarrow Y + B + R$ 。

并且, 如果你还记得, 积的箭头则是相反的——积的等价例子将是函数 $Y \times B \times R \rightarrow Y \times B$ 。

这一事实揭示了和 (sum) 和积 (product) 概念之间的深刻联系, 它们实际上是彼此的对立面。积是和的对立面, 而和是积的对立面。

在范畴论中，具有这种关系的概念被称为彼此的对偶 (dual)。因此，和 (sum) 和积 (product) 的概念是对偶的。这也是为什么和在范畴论环境中被称为逆积 (converse product)，或简称为上积 (coproduct)。这种命名约定用于范畴论中的所有对偶构造。

```
{% if site.distribution == 'print'%}
```

德摩根对偶 (De Morgan Duality)

现在让我们从逻辑的角度来看和积 (sum) 和积 (product) 的概念。我们提到过：

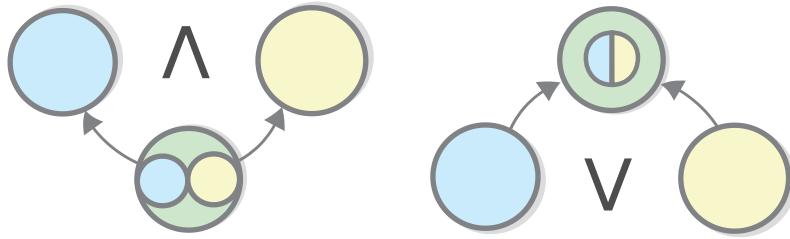
- 两个集合的积包含一个来自第一个集合的元素与一个来自第二个集合的元素。
- 两个集合的和包含一个来自第一个集合的元素或一个来自第二个集合的元素。

当我们将这些集合视为命题时，我们会发现积的概念 (\times) 与逻辑中的与关系完全一致 (记作 \wedge)。从这个角度来看，函数 $Y \times B \rightarrow Y$ 可以看作是逻辑推理规则中的一个实例，称为合取消去 (conjunction elimination) (也称为简化)，即 $Y \wedge B \rightarrow Y$ ，例如，如果你的头发是部分金色和部分棕色，那么它就是部分金色。

同样地，和 (+) 的概念与逻辑中的或关系一致 (记作 \vee)。从这个角度来看，函数 $Y \rightarrow Y + B$ 可以视为逻辑推理规则中的一个实例，称为析取引入 (disjunction introduction)，即 $Y \rightarrow Y \vee B$ 。例如，如果你的头发是金色的，那么它要么是金色的，要么是棕色的。

这意味着，与 和 或 的概念也是对偶的——这个思想由 19 世纪的数学家奥古斯都·德摩根 (Augustus De Morgan) 提出，并因此称为德摩根对偶 (De Morgan duality)，这是现代范畴论中对偶思想的前身。

这种对偶性被微妙地编码在逻辑符号中，表示与 和 或 (\wedge 和 \vee)——它们实际上是积和上积图表的简化版本 (虽然方向相反，但仍然成立)。



德摩根对偶

$$\neg(A \wedge B) = \neg A \vee \neg B$$

$$\neg(A \vee B) = \neg A \wedge \neg B$$

例如,你可以将第二个公式理解为,如果我的头发既不是金色也不是棕色,这意味着我的头发不是金色的并且不是棕色的,反之亦然(这种联系是双向的)。

现在我们将逐步解析这些公式,展示它们实际上是与 和 或 对偶性下的一个简单推论。

假设我们想找到“金色或棕色”的反命题。

$$A \vee B$$

我们首先要做的是,用它们的反命题替换构成它的陈述,这将使命题变为“不是金色或不是棕色”。

$$\neg A \vee \neg B$$

然而,这个命题显然不是“金色或棕色”的反命题(说我的头发不是金色的或不是棕色的,实际上允许它是金色的,也允许它是棕色的,只是不允许它同时是两者)。

那么我们遗漏了什么呢?很简单:我们替换了命题,但没有替换连接它们的运算符——对于两个命题,仍然是“或”。因此,我们必须用或的反命题替换它。正如我们之前所说,并且通过分析这个例子,你可以看到,这个运算符是与。因此,公式变为“不是金色并且不是棕色”。

$$\neg A \wedge \neg B$$

这个公式就是“金色与棕色”的反命题，也就是说，它等价于它的否定，正是德摩根第二定律所说的。

$$\neg(A \vee B) = \neg A \wedge \neg B$$

如果我们将整个公式“翻转”（我们可以这样做，而不改变单个命题的符号，因为它对所有命题都是有效的），我们得到了第一条定律。

$$\neg(A \wedge B) = \neg A \vee \neg B$$

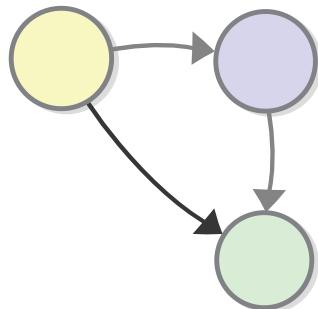
这可能会引发许多问题，我们有一整章关于逻辑的内容来解决这些问题。但在我们开始研究这些之前，我们需要看看什么是范畴（以及集合）。

{% endif %}

使用函数定义集合论的其余部分 (Defining the Rest of Set Theory Using Functions)

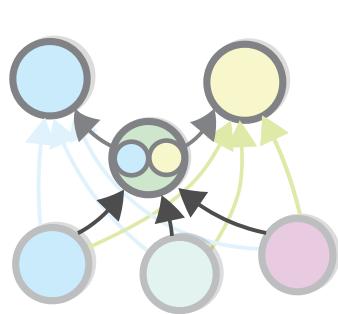
到目前为止, 我们已经看到了如何通过不查看集合元素并仅使用函数和外部图表来定义集合论的构造。

在第一章中, 我们使用如下图表定义了函数和函数的复合。

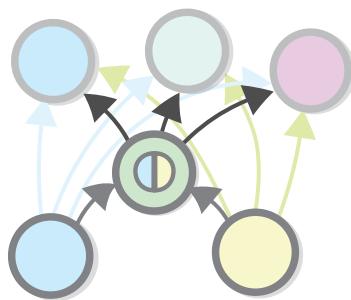


函数复合

现在我们也定义了积和和。



PRODUCT



COPRODUCT

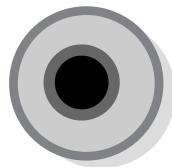
和积与积

更令人惊奇的是，我们可以基于函数的概念，定义整个集合论，这一发现归功于范畴论的先驱弗朗西斯·威廉·劳维尔 (Francis William Lawvere)。

用函数定义集合元素 (Defining Set Elements Using Functions)

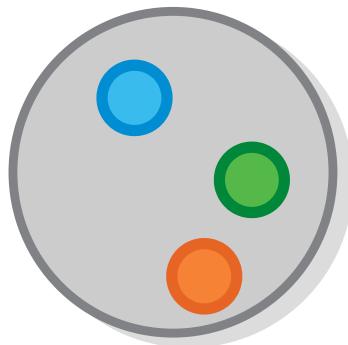
传统上，集合论中的一切都由两件事定义：集合和元素，所以如果我们要使用集合和函数来定义它，那么我们必须用函数定义集合元素的概念。

为此，我们将使用单例集合 (singleton set)。



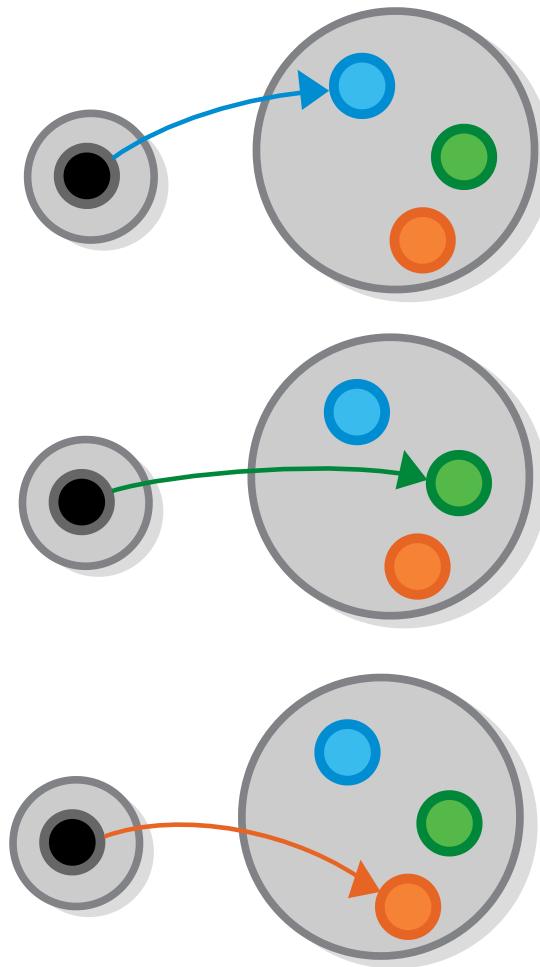
单例集合

好吧，让我们从描述一个随机集合开始。



一个三元素集合

然后我们检查从单例集合到这个随机集合的函数。



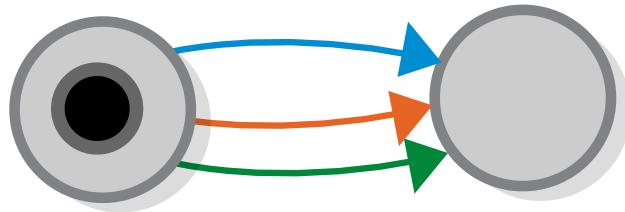
从单例集合到集合的函数

很容易看出,对于集合中的每个元素,将有一个唯一的函数,因此每个集合 X 的元素与一个函数 $1 \rightarrow X$ 是同构的。

因此,我们可以说所谓的集合“元素”就是从单例集合到它的函数。

用函数定义单例集合 (Defining the Singleton Set Using Functions)

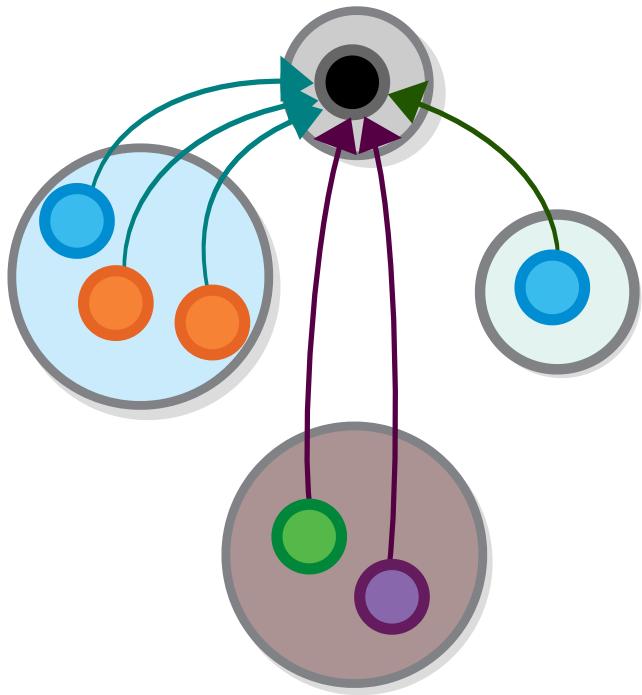
现在我们已经用函数定义了集合元素，我们可以尝试将集合的元素绘制成一个外部图表。



从单例集合到集合的函数

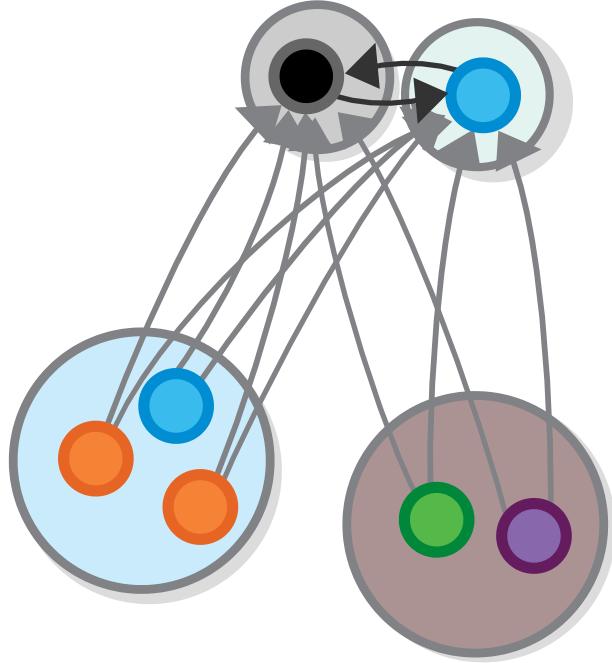
然而，我们的图表还没有完全外部化，因为它依赖于单例集合的概念，即具有一个元素的集合。此外，这使得整个定义是循环的，因为为了定义单元素集合的概念，我们必须已经定义了元素的概念。

为了避免这些困难，我们设计了一种仅使用函数来定义单例集合的方法。我们以与定义积和和相同的方式来做这件事——通过使用单例集合的唯一属性。具体来说，存在从任何其他集合到单例集合的唯一函数，即如果 1 是单例集合，那么我们有 $\forall X \exists! X \rightarrow 1$ 。



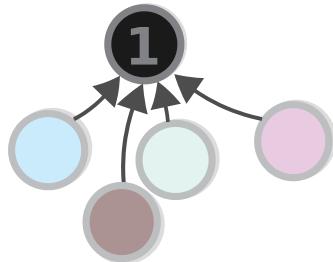
终对象

事实证明，这个属性唯一地定义了单例集合，即除了与单例集合同构的集合之外，没有其他集合具有这个属性。这只是因为，如果有两个集合具有这个属性，那么这两个集合之间也会存在唯一的态射 (morphism)，即它们将是同构的。更正式地说，如果我们有两个集合 X 和 Y ，使得 $\exists! X \rightarrow 1 \wedge \exists! Y \rightarrow 1$ ，那么我们也有 $X \cong Y$ 。



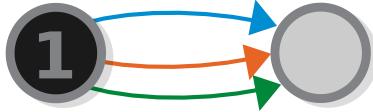
终对象

由于没有其他集合具有这个属性,我们可以使用它来定义单例集合,并且可以说如果我们有 $\forall X \exists! X \rightarrow 1$,那么 1 是单例集合。



终对象

通过这个定义,我们获得了单例集合的完全外部定义(同构意义下),因此获得了集合元素的定义——集合的元素只是从单例集合到该集合的函数。



从单例集合到集合的函数

请注意，从该属性可以推导出，单例集合只有一个元素。



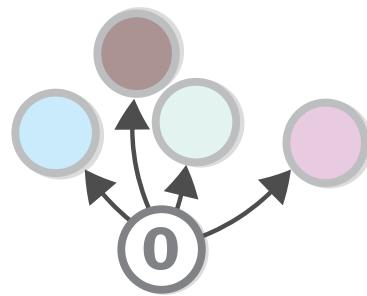
从单例集合到集合的函数

问题：为什么是这样呢（检查定义）？

用函数定义空集 (Defining the Empty Set Using Functions)

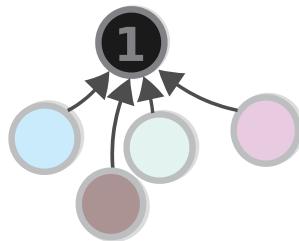
空集是没有元素的集合，但是我们如何在不引用元素的情况下表达这一点呢？

我们说过，存在从空集到任何其他集合的唯一函数。但反过来也成立：空集是唯一存在从它到任何其他集合的函数的集合。

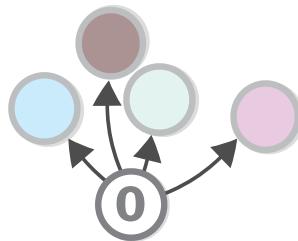


初对象

细心的读者会注意到，初对象和终对象的图表非常相似（是的，这两个概念当然是对偶的）。



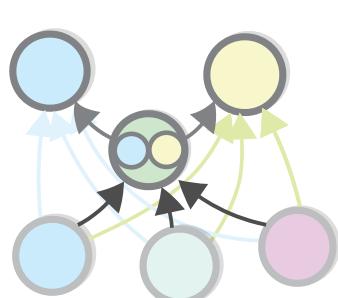
TERMINAL OBJECT



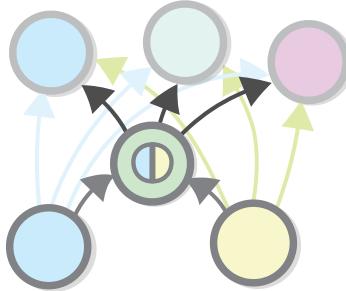
INITIAL OBJECT

初对象和终对象的对偶性

一些更加细心的读者可能还会注意到，积/上积图表与初/终对象图表也有相似之处。



PRODUCT



COPRODUCT

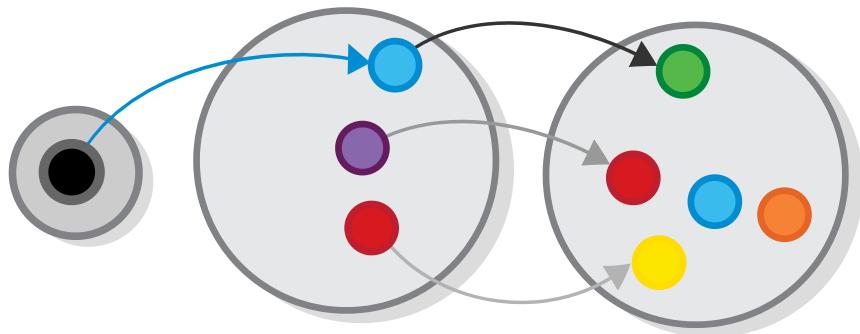
积与和积

(各位,保持冷静,你们观察力太强了——我们还有大约四章才能涉及到这些内容。)

函数应用 (Functional Application)

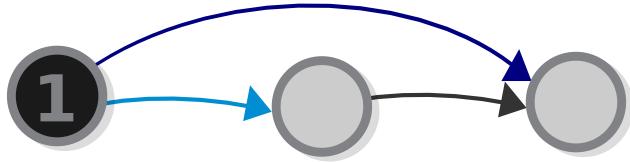
看到集合元素的函数定义后,我们可能会产生这样的疑问:如果元素由函数表示,那么我们如何将一个函数应用于一个集合的元素,以获得另一个集合的元素呢?

答案出乎意料的简单——为了将函数应用于一个集合,首先你必须选择集合的一个元素,而选择集合元素的行为与构造一个从单例集合到这个元素的函数是相同的。



函数应用——内部图表

然后,应用一个函数到一个元素,相当于将这个函数与我们想要应用的函数组合在一起。



函数应用——外部图表

结果是表示应用结果的函数。

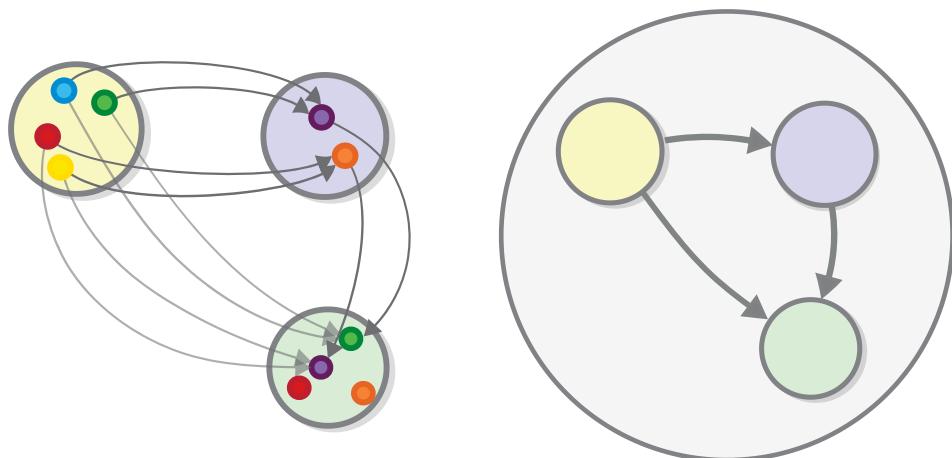
结论 (Conclusion)

未来，我们可能会讨论整个劳维尔的《集合范畴的初等理论》(Elementary Theory of the Category of Sets, ETCS)，并列出定义一个严谨的集合论所需的所有概念和公理，但现在这些已经足够让你理解主要思想：这些公理构成了一个集合论的定义，完全基于函数。这是一个关键思想，但还有更大的事情要讨论：因为它比传统定义更为普遍，这个新的定义也适用于不完全是集合但在某些方面类似于集合的对象。

你可以说它们适用于完全不同的对象类别(categories of objects)。

范畴论简要定义 (Category Theory — Brief Definition)

也许是时候看看什么是范畴了。我们将从一个简短的定义开始——范畴由对象 (objects) (一个例子是集合) 和从一个对象到另一个对象的态射 (morphisms) 组成 (可以视为函数)，而这些态射应该是可复合的 (composable)。我们可以对范畴说很多更多的内容，甚至可以提供一个形式化定义，但目前只需记住，集合是范畴的一个例子，而范畴对象类似于集合，除了我们看不到它们的元素。或者换句话说，范畴论的概念通过外部图表表示，而严格的集合论概念可以通过内部图表表示。



范畴论与集合论的比较

当我们处于集合的领域时，我们可以将每个集合视为个体元素的集合。在范畴论中，我们没有这样的概念。然而，去掉这个概念使我们能够以一种完全不同且更普遍的方式定义诸如和集合 (sum sets) 和积集合 (product sets) 之类的概念。此外，我们总是有办法“回到”集合论，使用上一节中的技巧。

但为什么我们想要一个更普遍的定义呢？这是因为，通过这种方式，我们可以使用我们的理论来描述非集合的对象。我们已经讨论过一个这样的对象——编程语言中的类型。还记得我们说过，编程类型（类）在某种程度上类似于集合，而编程方法类似于集合之间的函数，但它们并不完全相同？范畴论允许我们泛化这些……嗯，范畴（categories）的相似性。

范 畐 论 (Category Theory)	集 合 论 (Set Theory)	编 程 语 言 (Programming Languages)
范畴 (Category)	N/A	N/A
对 象 和 态 射 (Objects and Morphisms)	集 合 与 函 数 (Sets and Functions)	类 与 方 法 (Classes and Methods)
N/A	元 素 (Element)	对 象 (Object)

注意，通过集合论的视角看待的世界和通过范畴论视角看待的世界之间，存在一种奇怪的（但实际上完全合乎逻辑的）对称性（或许可以称为“反向对称性”）。

范 畐 论 (Category Theory)	集 合 论 (Set Theory)
范畴 (Category)	N/A
对 象 和 态 射 (Objects and Morphisms)	集 合 与 函 数 (Sets and Functions)
N/A	元 素 (Element)

通过切换到外部图表，我们失去了具体的视角（集合的元素），但我们获得了放大视角，能够看到我们之前被困住的整个宇宙。与整个集合领域可以视为一个范畴类似，编程语言也可以视为一个范畴。范畴的概念允许我们发现并分析这些和其他结构之间的相似性。

注意：在编程语言和范畴论中都使用了“对象”一词，但它们的含义完全不同。在范畴论中，范畴对象等同于编程语言理论中的类型或类。

集合 vs 范畴 (Sets Vs Categories)

在继续之前，我们先做一个说明：在过去的几段中，可能看起来范畴论和集合论在某种程度上是互相竞争的。也许这种观点在某种程度上是正确的，如果范畴论和集合论都是为了描述具体的现象，就像相对论和量子力学都试图解释物理世界一样。具体理论主要是作为对世界的描述来构建的，因此它们之间存在某种层次关系是合理的。

然而，抽象理论，如范畴论和集合论，更像是用于表达这种描述的语言——它们仍然可以相互联系，并且以不止一种方式确实联系在一起，但它们之间没有固有的层次关系，因此争论哪一个更基础或更普遍，实际上就是一个鸡生蛋还是蛋生鸡的问题，正如你将在下一章中看到的那样。

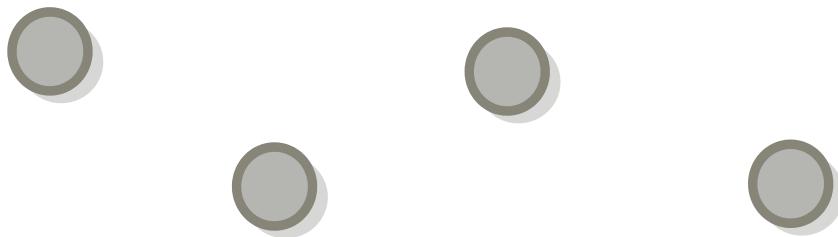
再次定义范畴 (Defining Categories Again)

“...通过忽略集合的所有元素，并仅关注集合的定义来处理它们。”
— 戴克斯特拉 (Dijkstra) (来自《论真正教授计算科学的残酷性》)

所有关于范畴论的书(包括这本)都从讨论集合论开始。然而回头看看，我真的不明白为什么会这样——大多数专注于某个主题的书通常不会在开始时引入完全不同的主题，即使这两个主题非常密切相关。

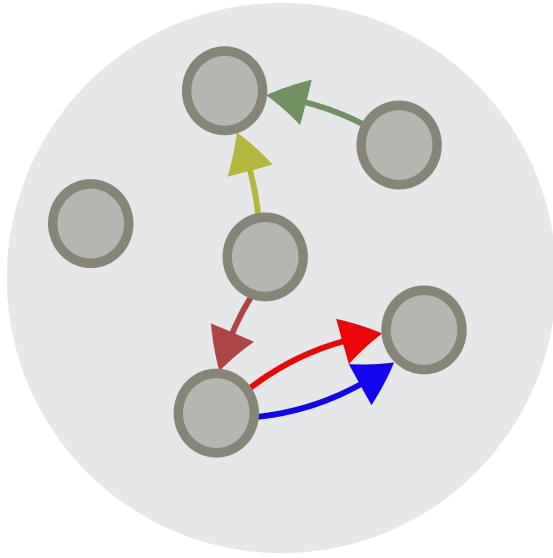
也许先讲集合是介绍范畴的最佳方式。或者，也许通过集合来介绍范畴是人们这样做的原因，只是因为每个人都这样做。但有一件事是确定的——我们不需要学习集合才能理解范畴。所以现在我想重新开始，并把范畴作为一个基础概念来讨论。让我们假装这是一本新书(我想知道是否可以将这本书献给不同的人)。

所以。范畴是对象 (things) 的集合，而这些“things”可以是你想要的任何东西。例如，考虑这些五颜六色的灰色球体：



球体

范畴由对象的集合以及一些连接对象的箭头组成。我们称这些箭头为态射 (morphisms)。

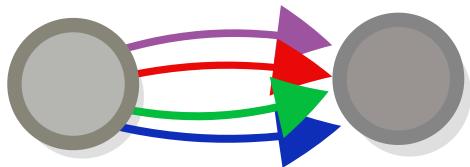


一个范畴

等一下，我们说过所有的集合构成一个范畴，但同时任何一个集合都可以被视为一个范畴本身（只是一个没有态射的范畴）。这是真的，这是范畴论一个非常典型的现象的例子——一种结构可以从多角度进行检查，并且可能在递归的方式中扮演许多不同的角色。

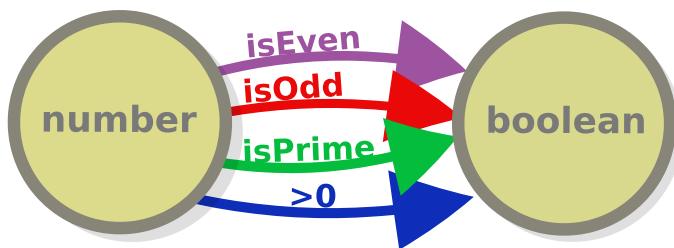
然而，这种比喻（一个没有态射的集合是一个范畴）并不是特别有用。不是因为它在任何方面都是不正确的，而是因为范畴论全都是关于态射的。如果集合论中的箭头仅仅是它们的源对象和目标对象之间的连接，那么在范畴论中，对象只是箭头连接其他对象的源和目标。因此，在上面的图表中，箭头而不是对象被着色：如果你问我，集合范畴应该被称为 **函数范畴**。

说到这里，注意范畴中的对象可以通过多个箭头相互连接，并且有相同的源对象和目标对象并不意味着这些箭头是等价的。



两个对象由多个箭头连接

为什么这是事实，如果我们回到集合论的角度来看，这是显而易见的（好吧，也许我们确实有时需要回到集合论）。例如，从数字到布尔值的函数是无限的，它们有相同的输入类型和输出类型（或者我们所说的类型签名），但这并不意味着它们在任何方面是等价的。

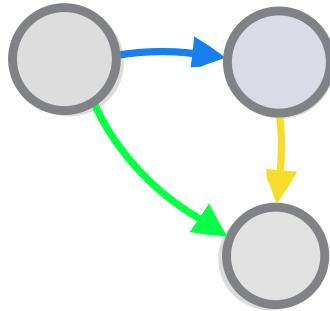


两个集合由多个函数连接

有一些类型的范畴，只允许在两个对象之间存在一个态射（或者每个方向一个），但我们以后会讨论它们。

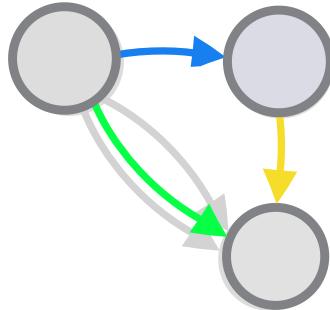
复合 (Composition)

一个结构被称为范畴的最重要要求是两个态射可以组成第三个，换句话说，态射是可复合的 (composable) —— 给定两个具有适当类型签名的连续箭头，我们可以画出第三个箭头，其等价于前两个函数的连续应用。



态射的复合

形式上,这个要求表明应该存在一个运算(用符号 \circ 表示),使得对于每两个函数 $g : A \rightarrow B$ 和 $f : B \rightarrow C$,存在一个函数 $(f \circ g) : A \rightarrow C$ 。



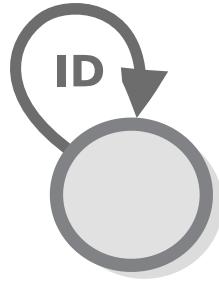
复合态射和额外态射

注意:请注意(如果你还没有注意到),函数复合是从右到左读取的。例如,应用 g 然后应用 f 被写为 $f \circ g$,而不是相反。(你可以把它看作是 $f(g(a))$ 的简写)。

同一律 (The Law of Identity)

在我们今天使用的标准阿拉伯数字出现之前,有罗马数字。罗马数字不好用,因为它们缺乏零的概念——一个表示数量缺失的数字,而任何缺乏这个简单概念的数字系统注定会非常有限。同样,在编程中,我们有多个表示值缺失的值。

范畴论中的零就是所谓的每个对象的“恒等态射”(identity morphism)。简而言之,这是一种不做任何事情的态射。



恒等态射(也可以是任何其他态射)

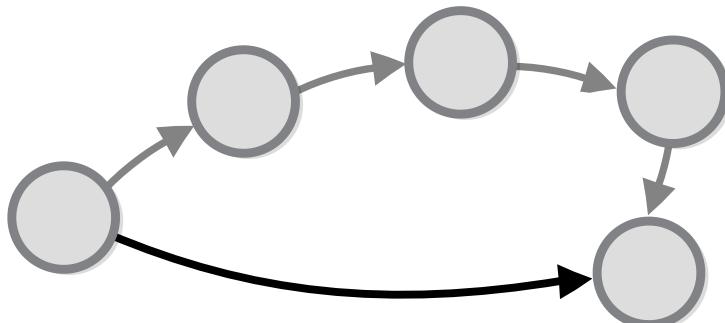
标记这个态射很重要, 因为从一个对象到同一对象可以有许多态射, 其中许多实际上是做事情的。例如, 数学处理的许多函数都以数字集合作为源和目标, 如 *negate*, *square*, *addone*, 而它们都不是恒等态射。

一个结构必须具有每个对象的恒等态射, 才能被称为范畴——这就是**恒等律**(law of identity)。

问题:在集合范畴中, 恒等态射是什么?

结合律 (The Law of Associativity)

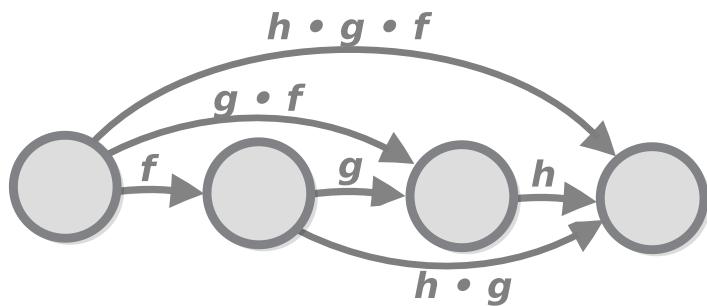
函数复合之所以特别, 不仅仅是因为你可以将两个具有适当签名的态射组合成第三个态射, 还因为你可以无限制地这样做, 即对于每个 n 个连续箭头, 每个箭头的源对象是上一个箭头的目标对象, 我们可以画出一个(唯一的)箭头, 它等价于所有 n 个箭头的连续应用。



具有多个对象的态射复合

让我们回到数学。仔细回顾上面的定义，我们可以看到它可以简化为以下公式的多次应用：给定 3 个对象和它们之间的两个态射 $f \circ g$ 和 h ，先将 h 与 g 组合，然后将结果与 f 组合，这应该与先将 h 组合到 g 和 f 的结果是相同的（或者简单地说 $(h \circ g) \circ f = h \circ (g \circ f)$ ）。

这个公式可以用下图表示，只有当公式为真时，图表才会交换（考虑到我们所有的范畴论图表都是交换的，我们可以说，在这种情况下，公式和图表是等价的）。



具有多个对象的复合态射

这个公式（以及图表）定义了一个称为结合性 (associativity) 的性质。结合性是函数复合真正被称为函数复合（以及一个范畴被真正称为范畴）的必要条件。它也是我们图表工作所必需的，因为图表只能表示结合结构（想象一下，如果上面的图表不交换，那会非常奇怪）。

结合性不仅仅与图表有关。例如，当我们使用公式表达关系时，结合性只是意味着公式中的括号无关紧要（正如定义 $(h \circ g) \circ f = h \circ (g \circ f)$ 所显示的那样）。

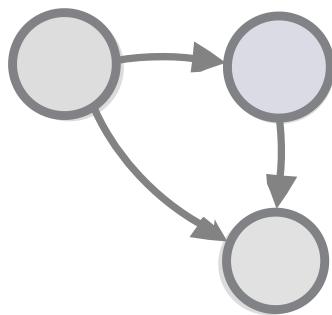
结合性不仅仅与范畴有关，它也是许多其他类型对象上的其他运算的一个性质。例如，如果我们看数字，我们可以看到乘法运算是结合的，例如 $(1 \times 2) \times 3 = 1 \times (2 \times 3)$ 。而除法不是 $(1/2)/3 \neq 1/(2/3)$ 。

这种组合无限多个事物的构建方法经常在编程中使用。要看到一些例子，你不需要看得太远，只需看看 Unix 中的管道操作符 (`|`)，它将一个程序的标准输出输入到另一个程序的标准输入中。如果你想看得更远，

注意有一种基于函数复合的编程范式, 称为“级联编程”, 它在 Forth 和 Factor 等语言中得到了应用。

交换图 (Commuting Diagrams)

上面的图表使用颜色来说明绿色的态射等同于其他两个态射(而不仅仅是一些无关的态射), 但实际上这种符号有点多余——画图表的唯一原因是表示等价路径。所有其他路径都只属于不同的图表。



态射复合——交换图

正如我们在上一章中简要提到的, 所有这样的图表(其中任何两个对象之间的两条路径是等价的)都被称为 **交换图表** (commutative diagrams) 或 **交换图**。本书中的所有图表(除非是错误构造的图表)都是交换的。

更正式地说, 一个交换图是一个图表, 其中给定两个对象 a 和 b 以及它们之间的两个态射序列, 我们可以说这些序列是等价的。

上面的图表是最简单的交换图之一。

总结 (A Summary)

为了以后参考, 让我们重新定义什么是范畴:

范畴是对象(我们可以将其视为点)和从一个对象到另一个对象的态射(或箭头)的集合, 其中: 1. 每个对象必须具有恒等态射。2. 应该有一种

方式将两个具有适当类型签名的态射复合成第三个态射，并且这种复合方式是结合的。

就这样。

```
{% if site.distribution == 'print'%}
```

附录：为什么范畴是这样的？(Addendum: Why are Categories Like That?)

为什么范畴是由这两个定律（而不是其他一两个、三个、四个等）定义的？从某个角度来看，答案似乎显而易见——我们研究范畴，因为它们有效，看看有多少应用。

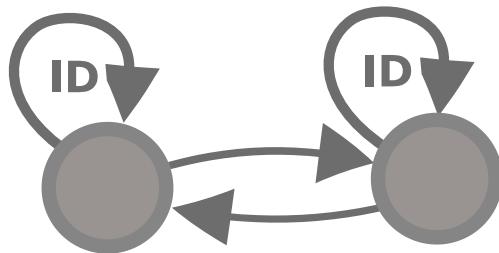
但同时，范畴论是一个抽象理论，所以它的所有内容都有点任意：你可以删除一个定律——你会得到另一个看起来类似范畴论的理论（尽管它在实践中可能完全不同（由于一个叫做“涌现现象”的现象））。或者你可以添加一个定律，得到另一个理论，所以如果这个特定的定律集比其他任何一个都更有效，那么这个事实就需要一个解释。虽然不是一个数学解释（例如，我们不能证明这个理论比其他某些理论更好），但仍然需要一个解释。接下来是我试图解释恒等性和结合性定律的原因。

恒等性和同构 (Identity and Isomorphisms)

恒等律是显而易见的一个要求。为什么需要有一个什么都不做的态射？这是因为态射是我们语言的基本构建块，我们需要恒等态射来能够正确地表达。例如，一旦我们定义了恒等态射的概念，我们就可以基于它定义一个范畴论的同构 (isomorphism) 概念（这很重要，因为同构的概念在范畴论中非常重要）：

正如我们在上一章中所说的，两个对象 (A 和 B) 之间的同构由两个态射组成——($A \rightarrow B$ 和 $B \rightarrow A$)，它们的复合等价于各自对象的恒等函数。形式上，如果存在态射 $f : A \rightarrow B$ 和 $g : B \rightarrow A$ 使得 $f \circ g = ID_B$ 且 $g \circ f = ID_A$ ，则对象 A 和 B 是同构的。

这里是同样的内容，用交换图表示。



同构

和前面一样，图表表达了与公式相同的（简单）事实，即从一个对象（ A 和 B 之一）到另一个对象，然后再回到起始对象与应用恒等态射相同，即什么都不做。

结合性和还原论 (Associativity and Reductionism)

“如果在某种灾难中，所有的科学知识都被摧毁，只传给下一代生物一个句子，那么哪句话能在最少的词中包含最多的信息？我认为是原子假说（或原子事实，或无论你怎么称呼它）——所有事物都是由原子构成的——小粒子，它们永恒地运动，相互吸引，距离稍远时靠近，但在被挤压在一起时排斥彼此。” — 理查德·费曼 (Richard Feynman)

结合性——它的含义是什么？为什么它存在？要回答这个问题，我们首先必须讨论另一个概念——还原论 (reductionism) 概念：

还原论是指一些更复杂的现象的行为可以用一些更简单和更基础的现象来理解，换句话说，事物随着它们变得“更小”而越来越简单（或者当它们在更低的层次上被观察时），例如，物质的行为可以通过研究其组成部分的行为来理解，即原子。

还原论是否普遍有效，即是否可以用更简单的事物解释一切（并制定出一种万物理论，将整个宇宙归结为几个非常简单的定律），这是一个可以争论到宇宙最终崩溃的问题。然而，确定的是，还原论支撑了我们所有

的理解，特别是在科学和数学领域——每个科学学科都有一套基础，用来解释一组更复杂的现象。例如，粒子物理学试图用一组基本粒子的行为来解释原子的行为，化学试图用化学元素的组成来解释各种化学物质的行为，等等。无法还原到某个学科基础的行为，显然就超出了该学科的范围（因此需要创建新的学科来解决它）。因此，如果这个原则如此重要，能够将其形式化会是很有益的，这就是我们现在将尝试做的。

交换律 (Commutativity)

还原论的一个陈述方式是说每个事物不过是其部分的总和。让我们尝试将其形式化。它意味着一组对象，无论以何种方式组合，都会始终产生相同的对象。

所以，如果我们有

$$A \square B \square C = D$$

我们也有

$$B \square A \square C = X$$

$$C \square A \square B = X$$

等等

或简单地

$$A \square B = B \square A$$

顺便说一下，这就是交换律 (commutativity) 的定义。

任务：如果我们的对象是集合，哪个集合运算可以表示这个总和？

结合性 (Associativity)

交换律只适用于顺序无关的情况，即当一个对象可以表示为无论以何种方式组合其部分的总和时。但是在许多情况下，一个对象要表示为其部分的总和，但只能按照特定方式组合。

在这些情况下，交换律不会成立，因为 A 可以与 B 组合得到 C ，这并不自动意味着 B 可以与 A 组合得到相同的结果（在函数的情况下，它们可能根本无法组合）。

但是，还原论的较弱版本在这种情况下仍然成立，即如果我们以特定顺序组合了一组对象，这组对象中的任何一对对象都可以随时被它们的组合结果所取代，即如果我们有

$$A \square B = D$$

和

$$B \square C = X$$

我们也有

$$(A \square B \square C) = D \square C = A \square X$$

或者简单地说

$$(A \square B) \square C = A \square (B \square C)$$

我认为这就是结合律的本质——通过缩小到你希望在特定时刻检查的部分来研究复杂现象，并单独观察它。

注意，结合律只允许在一维中组合事物。稍后我们将学习范畴论的扩展，它允许在二维中工作。

{%endif%}

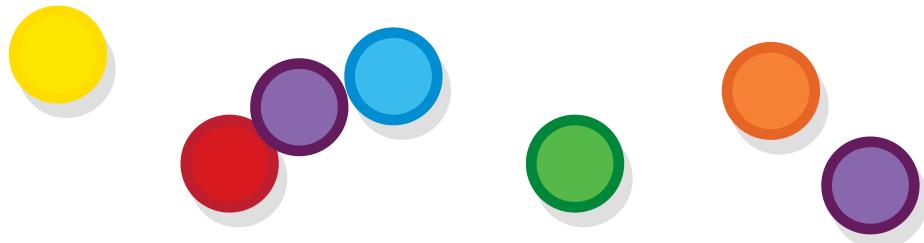
Monoids 等等 (Monoids etc)

既然我们已经讲完了范畴 (categories)，现在让我们看看一些同样有趣的结构——幺半群 (monoids)。与范畴一样，幺半群/群 (monoids/groups) 也是由元素集合及操作组成的抽象系统，用于操作这些元素。然而，这些操作与我们在范畴中使用的操作有所不同。让我们来看看。

什么是幺半群 (What are monoids)

幺半群比范畴简单得多。幺半群由一组元素 (称为幺半群的基础集合 (underlying set)) 和一个幺半群操作 (monoid operation) 定义——这是一个组合两个元素并产生同类第三个元素的规则。

让我们用我们熟悉的彩色球来说明。



Balls

我们可以通过定义一个“组合”两个球为一个的操作来基于这个集合定义一个幺半群。例如，这样的操作可以是将球的颜色混合，就像混合颜料一样。

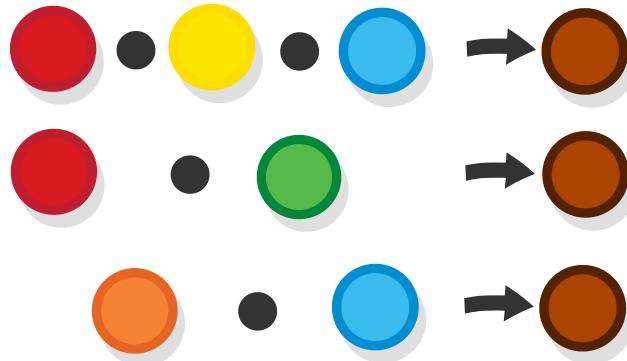


An operation for combining balls

你可能会想到定义类似操作的其他方法。这将帮助你意识到，可以从给定的元素集合中创建许多种不同的幺半群，也就是说，幺半群不仅仅是集合，它是集合与操作的结合。

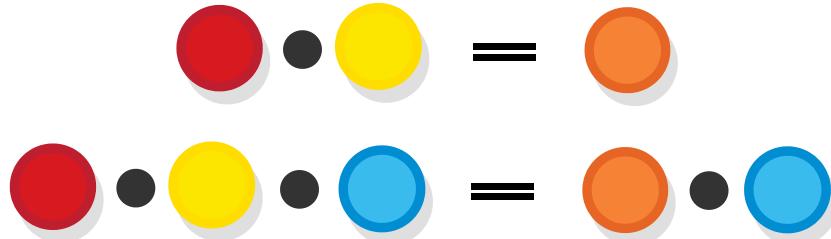
结合律 (Associativity)

幺半群的操作应该像函数组合一样具有结合性 (associative), 即以不同顺序应用它对相同数量的元素不应产生不同的结果。



Associativity in the color mixing operation

当一个操作具有结合性时, 这意味着我们可以对任何序列的项应用各种代数操作(或者换句话说, 应用等式推理), 例如我们可以用它由多个元素组成的集合替换任何一个元素, 或者添加一个在等式两边都存在的项, 并保持现有项的相等性。



Associativity in the color mixing operation

单位元 (The identity element)

实际上, 并非任何(结合的)组合元素的操作都能使球组成一个幺半群(它们可能组成半群(semigroup), 这也是一个主题, 但我们暂且不

谈）。要构成幺半群，一个集合必须有一个给定操作的单位元 (identity element)，这是你在集合论和范畴论中已经熟悉的概念——它是与任何其他元素组合时，返回该元素的元素（不是单位元，而是另一个元素）。简单来说， $x \cdot i = x$ 和 $i \cdot x = x$ 对于任何 x 都成立。

在我们的颜色混合幺半群中，白球（或者如果有的话，透明球）就是单位元。



The identity element of the color-mixing monoid

正如你可能还记得的，上章中函数组合也是结合的，并且也有一个单位元，所以你可能开始怀疑它以某种方式也构成了一个幺半群。确实如此，但这里有一个需要我们稍后讨论的细节。

基础幺半群 (Basic monoids)

为了保留悬念，在讨论幺半群与范畴之间的关系之前，我们先来看看一些简单的幺半群示例。

从数构成的幺半群 (Monoids from numbers)

数学不仅仅与数字有关，但数字确实会在大多数数学领域中出现，幺半群也不例外。当自然数集 \mathbb{N} 与我们熟悉的加法操作组合时，它构成了一个幺半群（传统上称为“在加法下”(under addition)）。这个幺半群记作 $\langle \mathbb{N}, + \rangle$ （通常，所有群体都用角括号括起集合和操作来表示）。

$$\mathbf{1} \ + \ \mathbf{1} \ = \ \mathbf{2}$$

The monoid of numbers under addition

如果你在课本中看到“ $1 + 1 = 2$ ”，你要么在阅读非常高级的内容，要么在阅读非常简单的内容，不过我不太确定这里到底属于哪种情况。

无论如何，自然数在乘法下也构成了一个幺半群。

$$\mathbf{1} \times \mathbf{1} = \mathbf{1}$$

The monoid of numbers under multiplication

问题 (Question): 这些幺半群的单位元是什么？

任务 (Task): 研究其他数学运算，并验证它们是否为幺半群。

布尔代数中的幺半群 (Monoids from boolean algebra)

思考我们所讨论的运算时，你可能会想起布尔运算中的与 (\wedge) 和或 (\vee)。这两者都形成幺半群，其作用于仅包含两个值的集合 $\{True, False\}$ 。

任务 (Task): 通过展开公式 $(A \wedge B) \wedge C = A \wedge (B \wedge C)$ 的所有可能值，证明 \wedge 是结合的。同样对或运算进行验证。

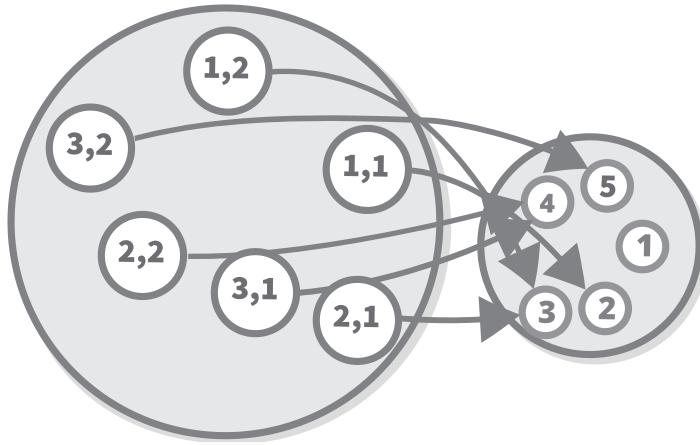
问题 (Question): “与”和“或”运算的单位元是什么？

幺半群运算的集合论定义 (Monoid operations in terms of set theory)

现在我们已经知道了什么是幺半群运算，甚至还看到了几个简单的例子。然而，我们还没有正式地定义幺半群规则/运算，即使用我们为其他内容定义时所用的集合论语言。我们能做到这一点吗？当然可以——一切都可以用集合来定义。

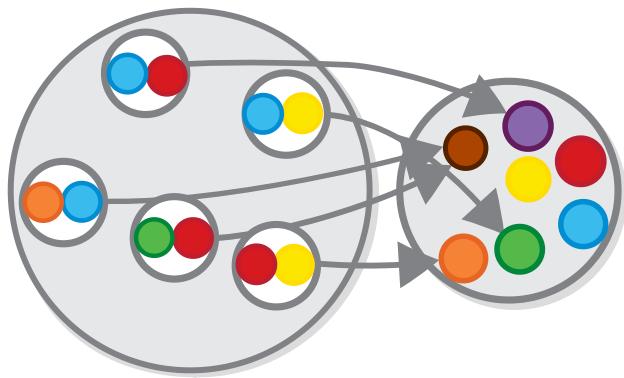
我们说过，幺半群由两个东西组成：一个集合（我们称之为 A ）和作用于该集合的幺半群操作。由于 A 已经在集合论中定义（因为它只是一个集合），我们只需定义幺半群操作。

定义这个操作并不难。实际上，我们已经为加法操作 + 做过类似的定义——在第2章中，我们说过“加法”可以用集合论表示为一个接受两个数的乘积并返回一个数的函数（形式上为 $+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ ）。



The plus operation as a function

其他任何幺半群操作也可以以相同的方式表示——作为一个函数，该函数从幺半群的集合中取一对元素并返回另一个幺半群元素。



The color-mixing operation as a function

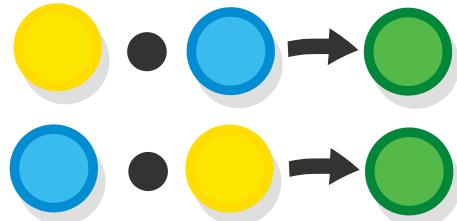
形式上,我们可以从任何集合 A 定义一个幺半群,通过定义一个具有类型签名 $A \times A \rightarrow A$ 的(结合的)函数。就是这样。准确来说,这是定义幺半群操作的一种方法。还有另一种方法,我们将在接下来看到。在那之前,让我们再研究一些范畴。

类似幺半群的其他对象 (Other monoid-like objects)

幺半群运算遵守两条定律——它们是结合的 (associative), 并且存在一个单位元 (identity element)。在某些情况下, 我们会遇到一些遵循其他定律的运算, 这些定律也同样有趣。向结合元素的方式施加更多 (或更少) 的规则, 会导致定义类似幺半群的结构。

交换幺半群 (Commutative (abelian) monoids)

查看幺半群定律和我们给出的示例时, 我们会发现所有这些例子都遵循一个我们没有明确说明的规则——应用操作的顺序对最终结果没有影响。



Commutative monoid operation

这样的操作 (对任何给定的元素集合进行组合, 无论先应用哪一个, 结果都是相同的) 称为交换的 (commutative) 操作。具有交换操作的幺半群称为交换幺半群 (commutative monoids)。

正如我们所说, 加法是交换的——无论我是先给你1个苹果然后再给你2个, 还是先给你2个再给你1个, 结果都是相同的。

$$\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$$

Commutative monoid operation

到目前为止, 我们研究的所有幺半群都是交换的。稍后我们会看到一些非交换的例子。

群 (Groups)

群是这样一种幺半群, 其中对于每个元素, 都存在一个所谓的“逆元”(inverse element), 当两个相继应用时, 它们会相互抵消。用平白的语言定义这些概念让你更能理解数学公式的优势——形式上我们说, 对于所有元素 x , 必须存在 x' 使得 $x \cdot x' = i$ (其中 i 是单位元)。

如果我们将幺半群视为建模一组(结合的)动作效果的手段, 我们则使用群来建模这些动作也是可逆的。

我们讨论过的一个既是幺半群又是群的例子是自然数集在加法下的集合。每个数的逆元是其相反数(正数的逆元是负数, 反之亦然)。上面的公式变成 $x + (-x) = 0$ 。

群的研究是一个比幺半群理论大得多的领域(甚至可能比范畴论本身还大)。其中一个最大的分支是“对称群”(symmetry groups)的研究, 我们将在接下来讨论。

总结 (Summary)

但是在此之前, 先做一个简单的总结——这些代数结构可以根据定义它们的定律汇总在以下表格中。

半群 (Semigroups) ✕ 半群 (Monoids)

结合律 (Associativity)	X
单位元 (Identity)	X
可逆性 (Invertibility)	

现在，让我们来看一下对称群 (symmetry groups)。

对称群及群分类 (Symmetry groups and group classifications)

几何图形的对称性 (symmetries) 群是一类有趣的幺半群/群。给定一个几何图形，对称性指的是一种动作，执行这种动作后图形不会发生位移（例如，它可以完美地适应在应用动作之前所适应的模具）。

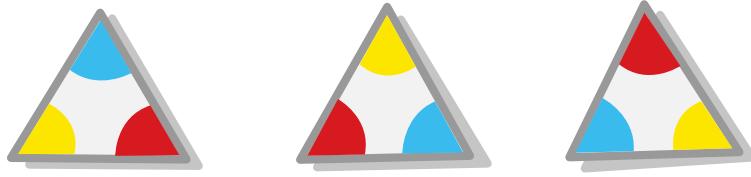
这次我们不会再使用球了，因为在对称性方面，球只有一个位置，因此只有一个动作——即恒等动作 (identity action)（顺便说一下，它是自己的逆元）。所以让我们使用这个三角形，就我们的目的而言，它和其他三角形没有区别（我们对三角形本身不感兴趣，而是对它的旋转感兴趣）。



A triangle

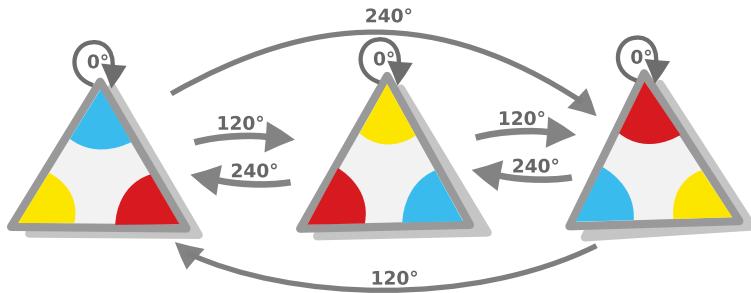
旋转群 (Groups of rotations)

首先让我们回顾一下我们如何旋转三角形的方式的群，即它的旋转群 (rotation group)。一个几何图形可以在不发生位移的情况下旋转的次数等于其边的数量，所以对于我们的三角形来说，有3个位置。



The group of rotations in a triangle

将点(在这里是三角形)连接起来表明,从任何状态到另一个状态的可能旋转方式只有两种——120度旋转(120-degree rotation)(即翻转一次三角形)和240度旋转(240-degree rotation)(即翻转两次,或者等价地,沿相反方向翻转一次)。加上零度旋转(恒等动作),总共有3种旋转方式(对象)。



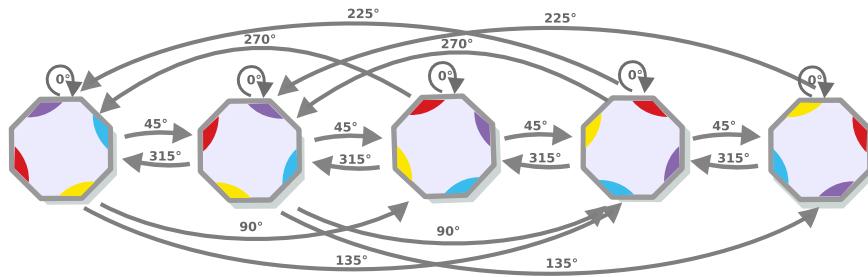
The group of rotations in a triangle

三角形的旋转构成了一个幺半群——旋转是对象(零度旋转是单位元),幺半群的操作是将两个旋转组合成一个,这就是先执行第一个旋转再执行第二个旋转的操作。

注意 (NB): 再次强调,群中的元素是旋转,而不是三角形本身,实际上这个群与三角形无关,正如我们稍后将看到的那样。

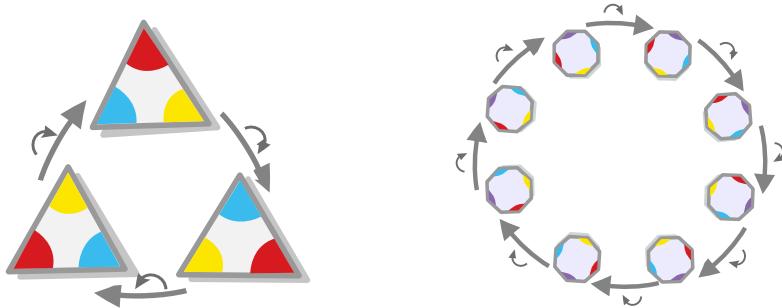
循环群/幺半群 (Cyclic groups/monoids)

更复杂的几何图形的旋转群的图表起初看起来相当杂乱。



The group of rotations in a more complex figure

但如果我们注意到以下几点，它会变得更容易理解：尽管我们的群有许多旋转，并且对于更多边的图形（如果我没记错的话，旋转的次数等于边数）有更多的旋转，所有这些旋转都可以归结为单一旋转的反复应用（例如，三角形的120度旋转和八边形的45度旋转）。让我们为这个旋转做个符号。

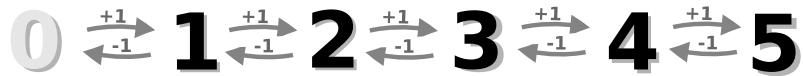


The group of rotations in a triangle

具有这样“主要”旋转的对称群，以及一般的幺半群和群，拥有一个对象，通过反复应用它可以生成所有其他对象，称为循环群（cyclic groups）。

这样的旋转称为群的生成元（generator）。

所有的旋转群都是循环群。另一个循环群的例子是自然数集在加法下的集合。这里我们可以使用 +1 或 -1 作为生成元。



The group of numbers under addition

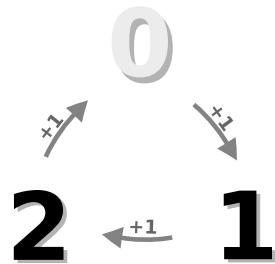
等等,既然没有循环,为什么这是循环群呢?因为整数是无限的循环群。

基于数的一个有限循环群的例子是模运算(modular arithmetic)下的自然数集(有时称为“时钟算术”(clock arithmetic))。模运算基于一个数,称为模数(modulus)(我们以12为例)。在其中,每个数都映射到该数与模数相加后取余数的结果。

例如: $1 \pmod{12} = 1$ (因为 $1/12 = 0$ 余 1) $2 \pmod{12} = 2$, 等等。

但是 $13 \pmod{12} = 1$ (因为 $13/12 = 1$ 余 1), $14 \pmod{12} = 2$, $15 \pmod{12} = 3$ 等等。

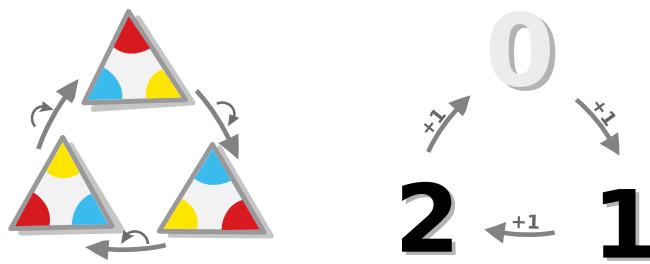
实际上,数“绕圈”形成一个与模数相同数量的元素的群。例如,模数为3的模运算的群表示有3个元素。



The group of numbers under addition

所有具有相同数量元素的循环群(或者它们是相同阶(same order)的)是同构的(细心的读者可能注意到我们还没有定义什么是群同构(group isomorphisms),更加细心的读者可能已经对它有了一个概念)。

例如, 三角形的旋转群与模 3 加法下的自然数集同构。



The group of numbers under addition

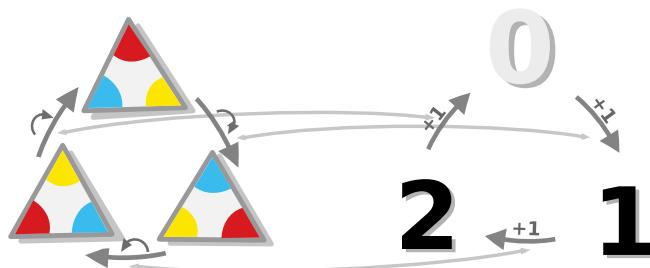
所有的循环群都是交换的 (commutative) (也称为“阿贝尔” (abelian) 群)。

任务 (Task): 证明除了 Z_3 之外, 没有其他具有3个对象的群。

有些阿贝尔群不是循环的, 但正如我们将在下文中看到的, 循环群和阿贝尔群的概念是密切相关的。

群同构 (Group isomorphisms)

我们已经提到过群同构, 但没有定义它们。现在让我们定义——两个群之间的同构是它们各自元素集合之间的同构 (f), 使得对于任何 a 和 b , 我们有 $f(a \square b) = f(a) \square f(b)$ 。直观上, 同构群的图表具有相同的结构。



Group isomorphism between different representations of S_3

在范畴论中，群论中的同构群被认为是同一个群的实例。例如上图所示的群称为 Z_3 。

有限群 (Finite groups)

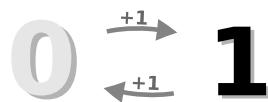
与集合一样，群论中的同构概念允许我们识别常见的有限群。

最小的群就是仅有一个元素的平凡群 (trivial group) Z_1 。

0

The smallest group

最小的非平凡群是 Z_2 ，它有两个元素。



The smallest non-trivial group

Z_2 也被称为布尔群 (boolean group)，因为它与 $True, False$ 集在否定运算下同构。

像 Z_3 一样， Z_1 和 Z_2 都是循环的。

群/幺半群的积 (Group/monoid products)

我们已经看到很多既是阿贝尔群也是循环群的例子，但我们还没有看到任何非循环的阿贝尔群。那么让我们看看它们的样子。这次，我们不再看单个例子，而是展示一种通过群积 (group product) 将循环群组合成非循环阿贝尔群的一般方法。

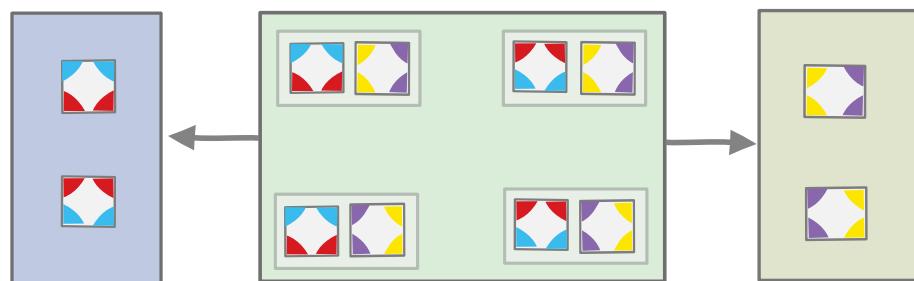
给定任意两个群，我们可以将它们组合成第三个群，该群由两个群的所有可能元素对组成，并包含它们所有操作的和。

让我们看看积的样子。取以下两个群（由于它们只有两个元素和一个操作，它们都与 Z_2 同构）。为了更容易想象它们，我们可以将第一个群看作基于图形的垂直反射，而第二个群则是水平反射。



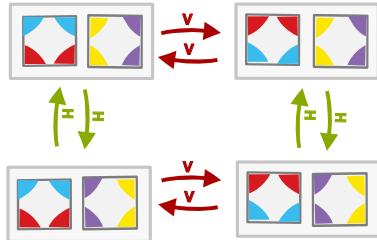
Two trivial groups

通过取第一个群的元素集合与第二个群的元素集合的笛卡尔积 (Cartesian product)，我们得到了新群的元素集合。



Two trivial groups

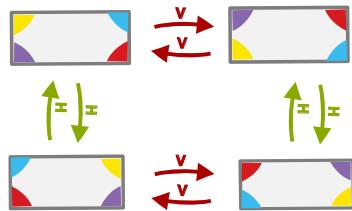
而积群的操作由第一个群的操作与第二个群的操作组成, 其中每个操作只作用于属于其对应群的元素, 而另一个元素保持不变。



Klein four

我们展示的两个群的积称为 **克莱因四元群** (Klein four-group), 它是最简单的**非循环阿贝尔** (non-cyclic abelian) 群。

克莱因四元群的另一种表示方式是非正方形矩形的对称群。



Klein four

任务 (Task): 证明这两种表示是同构的。

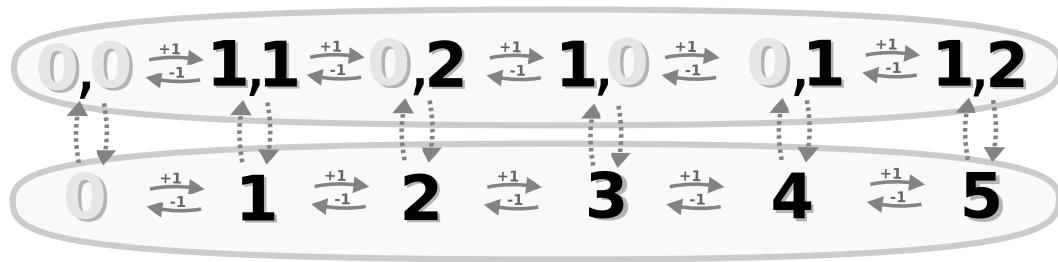
克莱因四元群是**非循环的**(因为它有两个生成元)——垂直旋转和水平旋转。然而, 它仍然是**阿贝尔的**, 因为操作顺序对最终结果没有影响。实际上, 克莱因四元群是**最小的非循环群**。

循环积群 (Cyclic product groups)

当组成积的群的元素数量(即它们的阶(orders))不是互质(relatively prime)时,积群是非循环的。

如果两个群的阶不是互质的(例如2和2,它们都能被2整除,就像构成克莱因四元群的群一样),那么即使这两个群都是循环的,并且每个群只有一个生成元,它们的积也会有两个生成元。

如果你将两个阶是互质的群组合在一起(例如2和3),所得群将与同阶的循环群同构,因为 Z_3 与 Z_2 的积与 Z_6 群同构($Z_3 \times Z_2 \cong Z_6$)。



Chinese remainder theorem

这是一个古老结果的推论,称为中国剩余定理(Chinese Remainder theorem)。

阿贝尔积群(Abelian product groups)

当组成积的群是阿贝尔的时,积群也是阿贝尔的。我们可以通过注意到,虽然生成元不止一个,但每个生成元只作用于它自己群的部分,因此它们之间不会相互干扰,从而看到这一点。

有限阿贝尔群基本定理(Fundamental theorem of Finite Abelian groups)

积提供了一种从循环群创建非循环阿贝尔群的方法——通过将两个或多个循环群相乘。有限阿贝尔群基本定理是一个结果,告诉我们这是生成非循环阿贝尔群的唯一方法,即

所有阿贝尔群要么是循环的，要么是循环群的积。

我们可以利用这一定律来直观理解阿贝尔群的本质，也可以用它来检验某个给定的群是否可以分解为更多基本群的积。

```
{% if site.distribution == 'print' %}
```

颜色混合幺半群作为积 (Color-mixing monoid as a product)

为了看看我们如何使用这个定理，让我们重温一下之前看到的颜色混合幺半群。



color-mixing group

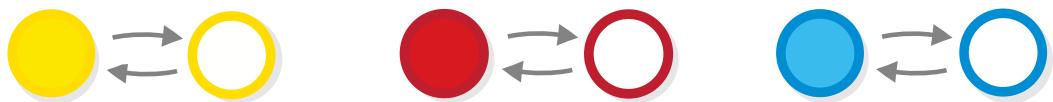
由于不存在一个可以通过与自身混合生成所有其他颜色的颜色，颜色混合幺半群是非循环的。然而，颜色混合幺半群是阿贝尔的。因此，根据有限阿贝尔群定理（该定理对幺半群同样适用），颜色混合幺半群必须（同构于）一个积。

而且，找到组成它的幺半群并不难——虽然没有一种颜色可以生成所有其他颜色，但有三种颜色可以做到——即三原色。这一观察使我们得出结论，颜色混合幺半群可以表示为三个幺半群的积，对应于三种原色。



color-mixing group as a product

你可以将每个颜色幺半群视为一个布尔幺半群, 只有两种状态(有颜色和无颜色)。



Cyclic groups, forming the color-mixing group

或者, 替代地, 你可以将其视为具有多个状态, 表示不同的着色层次。



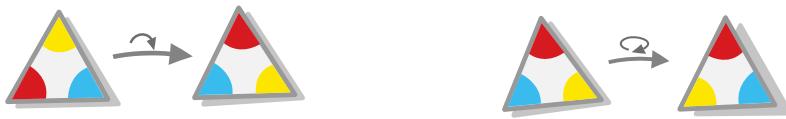
Color-shading cyclic group

在这两种情况下, 幺半群都是循环的。

{%endif%}

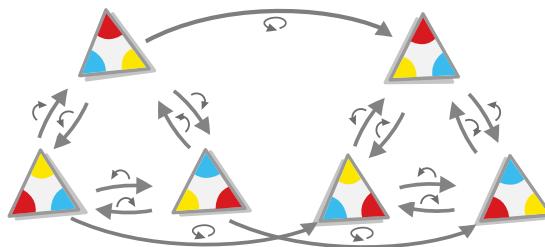
二面体群 (Dihedral groups)

现在，我们终于要研究一个非交换群 (non-commutative group)——一个几何图形的旋转和反射 (rotations and reflections) 的群。与上一个群类似，但这里除了我们已经看到的旋转操作 (及其复合操作) 之外，还有一个操作是垂直翻转图形，这个操作会使图形变为其镜像：



Reflection of a triangle

这两个操作及其复合结果形成一个称为 Dih_3 的群，它是非阿贝尔的 (而且是最小的非阿贝尔群)。



The group of rotations and reflections in a triangle

任务 (Task): 证明该群确实是非阿贝尔的。

问题 (Question): 除了有两个主要操作外，是什么决定了该群及其他任何群为非阿贝尔的？

表示任何二维形状的旋转和反射集的群称为二面体群 (dihedral groups)。

幺半群/群的范畴化视角 (Groups/monoids categorically)

我们首先定义了幺半群 (monoid) 为一组可组合的元素。然后我们看到对于某些群 (groups)，如对称群 (groups of symmetries) 和旋转群 (rotations)，这些元素可以视为动作。事实上，这对于其他所有群也同样适用。例如，我们颜色混合幺半群 (color-blending monoid) 中的红球可以看作是将红色加入混合物的动作，数字 2 在加法幺半群中可以看作是操作 $+2$ 等等。这一观察引出了幺半群和群的范畴化视角。

柯里化 (Currying)

当我们定义幺半群时，看到它们的操作是双参数函数。我们使用集合论 (set theory) 引入了一种表示这些函数的方法——通过使用积将两个参数合并。也就是说，我们展示了一个接受两个参数的函数（比如 A 和 B ）并将它们映射到某个结果 (C)，可以视为从这两个参数集合的积映射到结果。所以 $A \times B \rightarrow C$ 。

然而，这并不是集合论表示多参数函数的唯一方式——还有另一种同样有趣的方式，它不依赖于任何数据结构，而只依赖于函数：这种方式是有一个函数，它将第一个参数（即 A ）映射到另一个函数，该函数将第二个参数映射到最终结果（即 $B \rightarrow C$ ）。所以 $A \rightarrow B \rightarrow C$ 。

将一个接受一对对象的函数转换为一个接受一个对象并返回一个接受另一个对象的函数的过程称为柯里化 (currying)。它通过高阶函数来实现。下面是这种函数的实现方式。

```
const curry = <A, B, C> (f: (a:A, b:B) => C) => (a:A) => (b:B) => f(a, b)
```

同样重要的是反向函数，它将柯里化的函数映射为多参数函数，称为反柯里化 (uncurry)。

```
const uncurry = <A, B, C> (f:(a:A) => (b:B) => C) => (a:A, b:B) => f(a)(b)
```

关于这两个函数还有很多可以说的，首先是它们的存在引发了范畴论中积 (product) 和态射 (morphism) 概念之间的一种有趣关系，称为伴随 (adjunction)。但我们将再后面讨论这一点。目前，我们关心的是这两种函数表示是同构的，形式上为 $A \times B \rightarrow C \cong A \rightarrow B \rightarrow C$ 。

顺便说一下，这种同构也可以用编程来表示。它等价于以下函数对于任何参数总是返回 `true` 的声明：

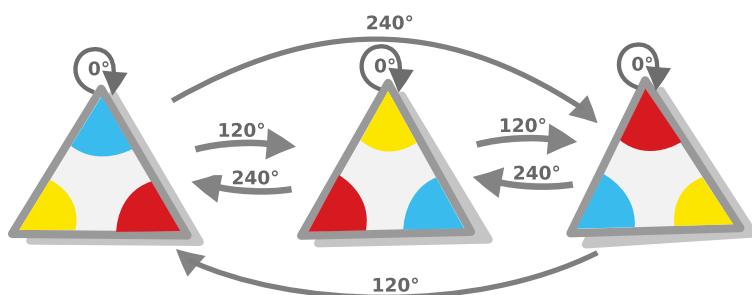
```
(...args) => uncurry(curry(f(...args))) === f(...args)
```

这是同构的一部分，另一部分是柯里化函数的等价函数。

任务 (Task): 编写同构的另一部分。

幺半群元素作为函数/置换 (Monoid elements as functions/permuations)

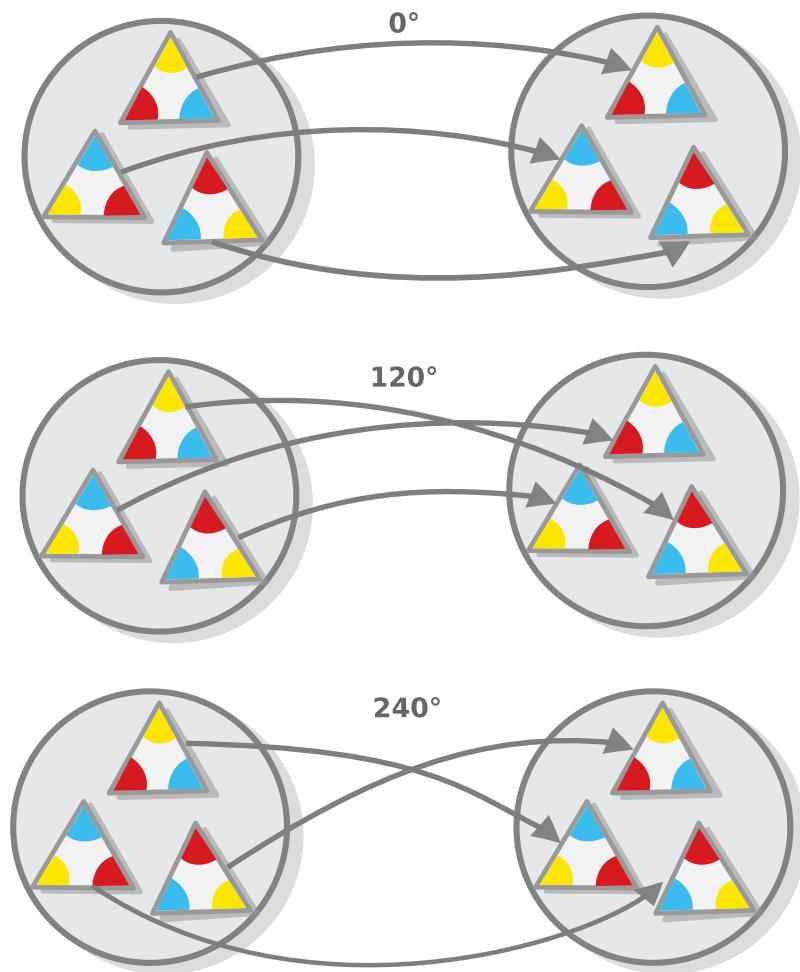
让我们回头看看我们到目前为止讨论的幺半群/群，结合我们学到的知识。我们一开始将群操作表示为一个对的函数。例如，对称群的操作（以 Z_3 为例）是将两个旋转转换为另一个旋转的动作。



The group of rotations in a triangle - group notation

通过柯里化，我们可以将一个给定群/幺半群的元素表示为函数，并将群操作本身表示为函数的组合。例如， Z_3 的三个元素可以看作是从一个包

含三个元素的集合到自身的三个双射函数(在群论背景下,这种函数称为置换(permutations))。



The group of rotations in a triangle - set notation

我们也可以对加法幺半群进行同样的操作——数字可以不被视为数量(例如两个苹果、两个橙子等),而是操作(例如作为将一个给定数量加上2的动作)。

形式上,加法幺半群的操作可以有如下类型签名:

$$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

由于我们上面展示的同构关系,这个函数等价于如下函数:

$$+ : \mathbb{Z} \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z})$$

当我们将在幺半群的一个元素(例如 2)应用于该函数时,结果是将 2 加到一个给定数上的函数 $+2$ 。

$$+2 : \mathbb{Z} \rightarrow \mathbb{Z}$$

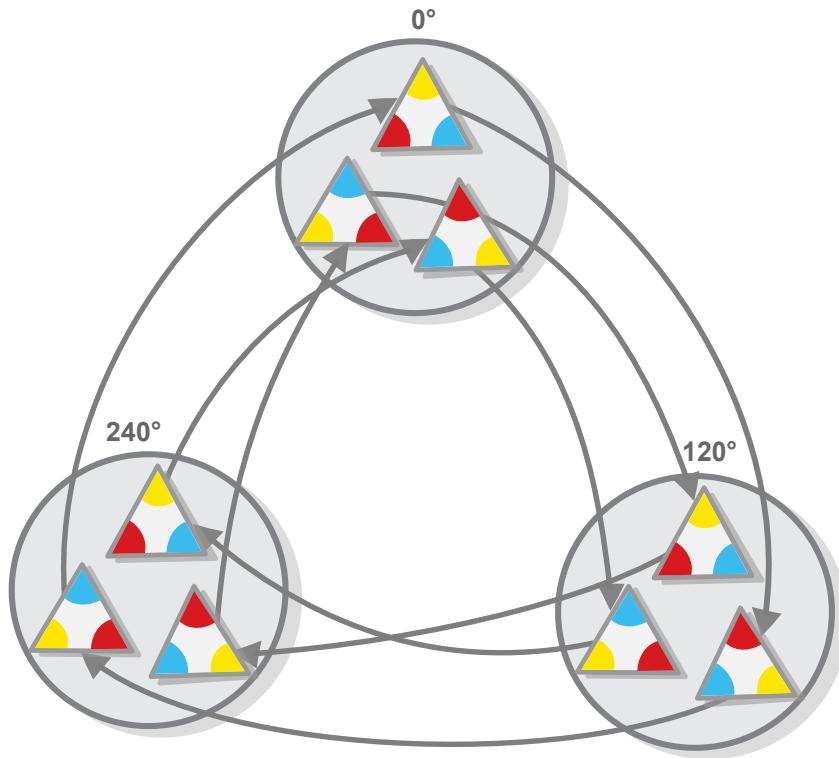
由于幺半群操作在上下文中是给定的,因此我们可以将元素 2 和函数 $+2$ 在幺半群的上下文中视为等价。

$$2 \cong +2$$

换句话说,除了将幺半群的元素表示为对象,我们也可以将它们表示为函数。

幺半群操作作为函数组合 (Monoid operations as functional composition)

表示幺半群元素的函数具有相同的源和目标,或者如我们所说的相同的类型签名(形式上,它们属于类型 $A \rightarrow A$,其中 A 为某个集合)。因此,它们可以通过函数组合(functional composition)相互组合,生成同样类型的签名的函数。



The group of rotations in a triangle - set notation

同样,这也适用于加法幺半群——数字函数可以通过函数组合进行组合。

$$+2 \square +3 \cong +5$$

因此,表示幺半群元素的函数也构成了一个幺半群,在函数组合操作下(表示群元素的函数在函数组合下也构成了一个群)。

问题 (Question): 哪些是函数群的单位元?

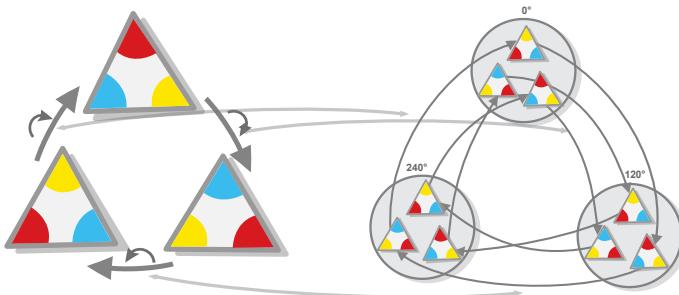
任务 (Task): 证明表示逆群元素的函数也是逆函数。

凯莱定理 (Cayley's theorem)

一旦我们学会如何通过柯里化将任何幺半群的元素表示为置换, 它们也形成了一个幺半群, 这样就不奇怪了, 这个构造的置换幺半群与原始幺半群同构(即从其构造的幺半群——这是一个称为凯莱定理的结果:

任何群都同构于一个置换群。

形式上, 如果我们用 Perm 表示置换群, 则对于任何集合 A , 有 $\text{Perm}(A) \cong A^A$ 。



The group of rotations in a triangle — set notation and normal notation

换句话说, 将一个群的元素表示为置换实际上产生了该幺半群的一个表示 (有时称为其标准表示 (standard representation))。

凯莱定理也许看起来并不那么令人印象深刻, 但这恰恰说明了它作为一个结果有多么具有影响力。

```
{% if site.distribution == 'print' %}
```

插曲: 对称群 (Symmetric groups)

首先需要知道的是, 对称群不等同于对称性群体 (symmetry groups)。当我们明确了这一点后, 我们可以理解它们实际是什么: 给定一个自然数 n , n 的对称群, 记作 S_n (n 度的对称群), 是一个集合所有可能的置换组成的群。这样的群的元素数等于 $1 \times 2 \times 3 \dots \times n$ 或 $n!$ (即 n 的阶乘)。

例如, S_1 表示一元素集的置换群只有一个元素 (因为一元素集除了恒等函数之外没有其他函数)。



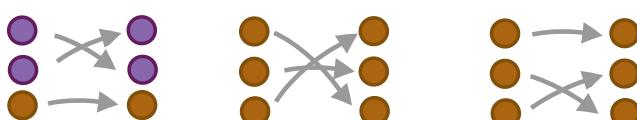
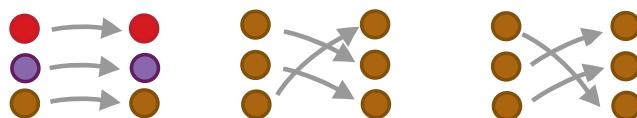
The S_1 symmetric group

S_2 有 $1 \times 2 = 2$ 个元素(顺便说一下, 颜色用于直观表示为什么 n 元素集的置换数为 $n!$)。



The S_2 symmetric group

到了 S_3 , 我们已经感受到了指数增长(甚至比指数增长更快!)的力量——它有 $1 \times 2 \times 3 = 6$ 个元素。



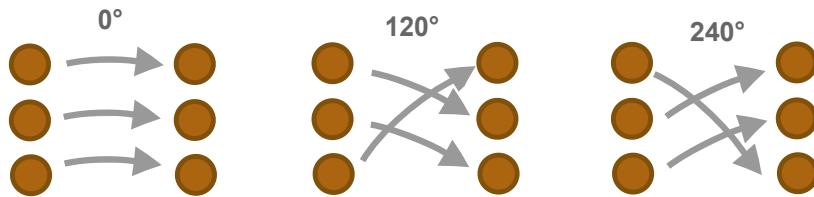
The S_3 symmetric group

每个对

称群 S_n 都包含所有 n 阶的群——这是因为(如我们在上一节所见)每个具有 n 个元素的群都同构于 n 元素集上的一个置换集, 而 S_n 包含了

所有此类存在的置换。

以下是一些例子：- S_1 同构于 Z_1 , 即平凡群 (trivial group), 而 S_2 同构于 Z_2 , 即布尔群 (boolean group) (但其他对称群不与循环群同构)。- S_3 的前三个置换同构于 Z_3 。



The S_3 symmetric group

- S_3 也同构于 Dih_3 (但其他对称群不与二面体群同构)。

基于这一见解, 我们可以这样表述凯莱定理：

所有群都同构于对称群的子群。

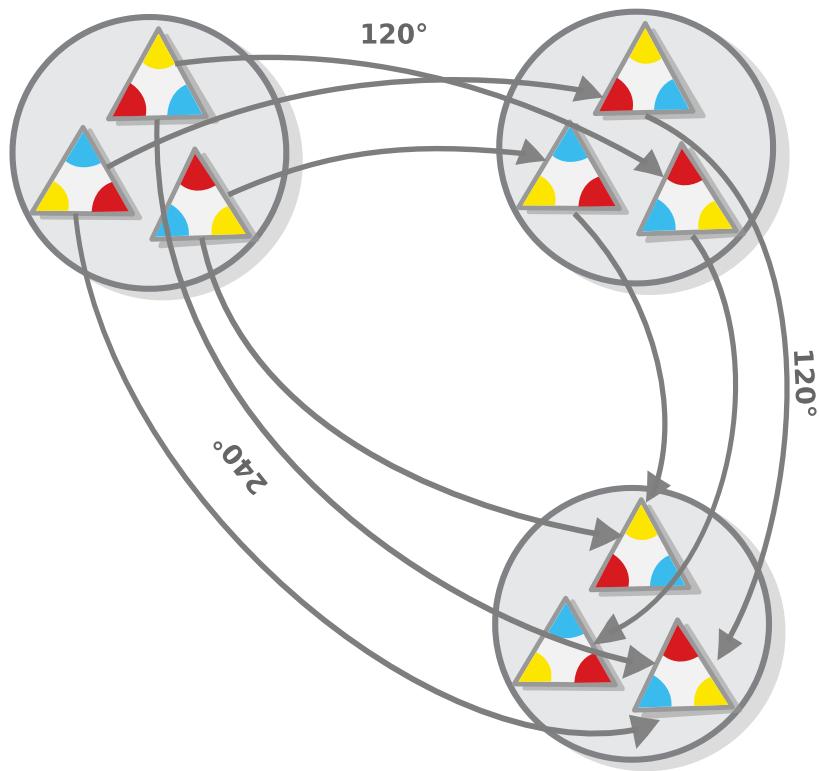
任务 (Task): 证明两者是如何等价的。

有趣的事實: 群论研究实际上始于对称群的研究, 因此该定理实际上是我們现在所知并热爱的群的标准定义出現的先决条件(好吧, 至少我喜欢它)——它提供了证明, 该定义所描述的概念与对称群已有的概念是等价的。

{% endif %}

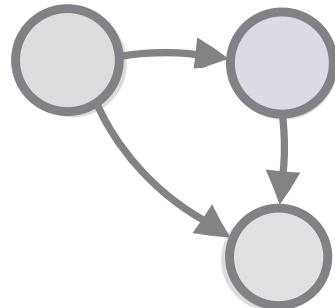
幺半群作为范畴 (Monoids as categories)

我们看到, 将幺半群的元素转换为动作/函数, 可以在集合和函数的上下文中准确地表示幺半群。



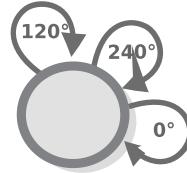
The group of rotations in a triangle - set notation and normal notation

然而,似乎这种表示中的集合部分有点多余——你到处都有相同的集合——所以,如果我们能简化它,那就好了。而我们可以通过将其描绘为一个外部(范畴的)图来做到这一点。



The group of rotations in a triangle - categorical notation

但是等一下,如果幺半群的集合对应于范畴论中的对象,那么对应的范畴将只有一个对象。因此,正确的表示将仅涉及一个点,从中发出的箭头和回到该点的箭头。



The group of rotations in a triangle - categorical notation

不同幺半群之间的唯一区别是它们拥有的态射数量及其之间的关系。

从范畴论的角度看,这种表示的直觉由幺半群和群的封闭性 (closure) 定律所体现,而范畴没有该定律——它规定无论如何组合两个对象,结果总是相同的对象。例如,不管你如何旋转一个三角形,你仍然得到一个三角形。

范畴 (Categories) 幺半群 (Monoids)

结合性 (Associativity)	X
单位元 (Identity)	X
可逆性 (Invertibility)	
封闭性 (Closure)	X

当我们将幺半群视为范畴时,这条定律表明该范畴中的所有态射都应该来自同一个对象并指向同一个对象——一个幺半群,无论是什么幺半群,都可以看作是具有一个对象的范畴。反之亦然:任何具有一个对象的范畴都可以看作是一个幺半群。

让我们通过回顾第2章中的范畴定义来详细说明这一想法。

范畴是对象(我们可以将其视为点)和态射(箭头)的集合,态射从一个对象指向另一个对象,其中: 1. 每个对象必须有一个恒等态

射。2. 必须有一种方法将两个具有适当类型签名的态射组合成一个第三个态射，并且这种组合是结合的。

除了幺半群的对象在范畴中是态射这一有些令人困惑的事实外，这正是幺半群的描述。

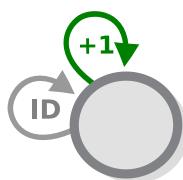
范畴对于每个对象都有一个恒等态射，因此对于只有一个对象的范畴，也应该有一个唯一的恒等态射。而幺半群确实有一个恒等对象，从范畴论的角度来看，它对应于那个恒等态射。

范畴提供了一种将两个具有适当类型签名的态射组合在一起的方法，对于只有一个对象的范畴，这意味着所有态射都应该可以组合。而幺半群操作正是这样做的——给定任意两个对象（或者说，两个态射，如果我们使用范畴术语），它会生成第三个态射。

哲学上，将幺半群定义为一个单对象范畴意味着它对应于这样一种观点，即幺半群是对一组（结合的）动作如何在给定对象上执行并改变其状态的建模。假设对象的状态仅由执行的动作决定，我们可以将其排除在外，专注于动作如何组合。通常情况下，这些动作（和元素）可以是任何东西，从混合颜色、给定数量的事物增加等。

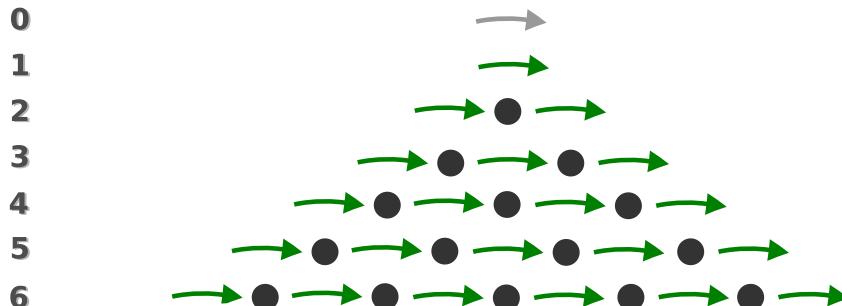
群/幺半群的表示 (Group/monoid presentations)

当我们把循环群/幺半群视为范畴时，我们会看到它们对应于具有一个对象和一个态射（如我们所说的生成元）的范畴，以及生成元与其自身组合时生成的态射。事实上，无限循环幺半群（同构于自然数），可以通过这个简单定义完全描述。



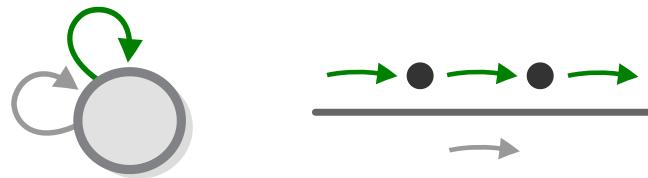
Presentation of an infinite cyclic monoid

这是因为生成元一次又一次的应用会产生无限循环群的所有元素。具体来说，如果我们将生成元视为操作 $+1$ ，则我们得到自然数。



Presentation of an infinite cyclic monoid

有限循环群/幺半群也是如此，只是它们的定义包含一条附加规则，规定一旦你将生成元与其自身组合了 n 次，就会得到恒等态射。对于循环群 Z_3 （可以看作是三角形旋转群），该规则规定生成元与其自身组合三次后得到恒等态射。



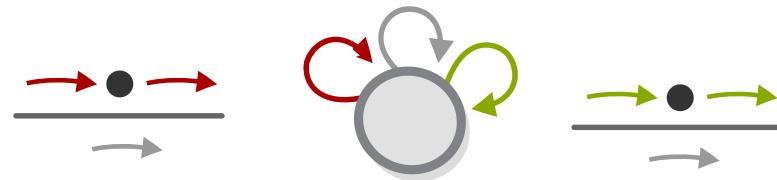
Presentation of a finite cyclic monoid

将群生成元与其自身组合，然后应用该规则，得到 Z_3 的三个态射。



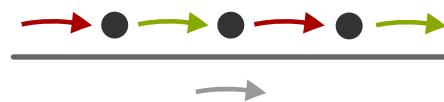
Presentation of a finite cyclic monoid

我们也可以用这种方式表示积群。让我们以克莱因四元群为例，克莱因四元群有两个生成元，它继承自组成它的群（我们视为非正方形矩形的垂直和水平旋转），每个生成元都有一个规则。



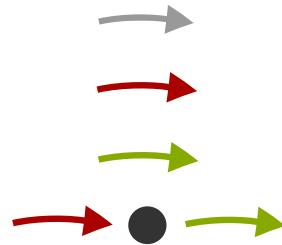
Presentation of Klein four

为了使表示完整，我们添加了组合这两个生成元的规则。



Presentation of Klein four - third law

然后，如果我们开始应用这两个生成元并遵循这些规则，我们就得到了四个元素。



The elements of Klein four

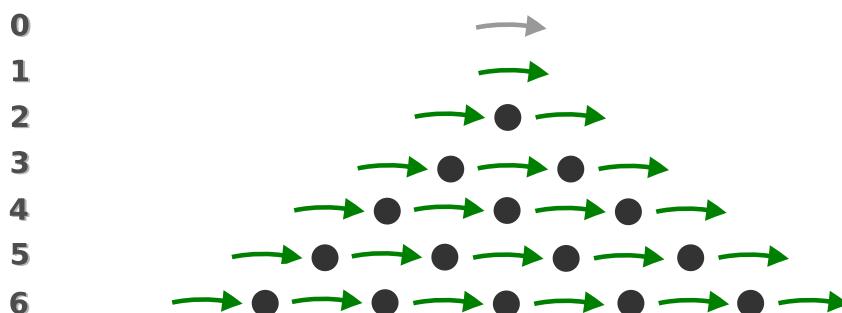
定义给定群的生成元和规则集合称为群的表示 (presentation of a group)。每个群都有一个表示。

```
{% if site.distribution == 'print' %}
```

插曲：自由幺半群 (Free monoids)

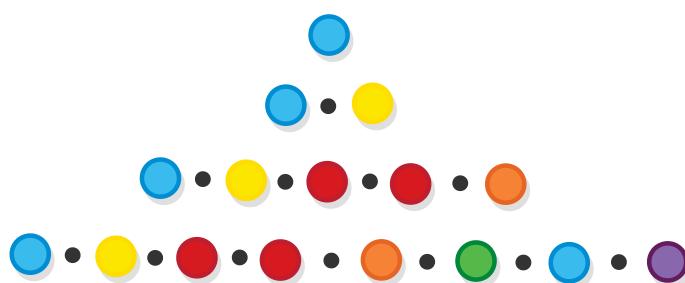
我们看到，选择不同的规则集合会产生不同类型的幺半群。那么如果我们不选择任何规则呢？这些幺半群（我们根据选择的集合得到不同的幺半群）称为自由幺半群 (free monoids)（“自由”一词在这里的意思是，一旦你有了集合，你可以“免费”将其升级为幺半群，即无需定义其他任何内容）。

如果你重温前一节，你会注意到我们已经看到过一个这样的幺半群。具有一个生成元的自由幺半群与整数幺半群同构。



The free monoid with one generator

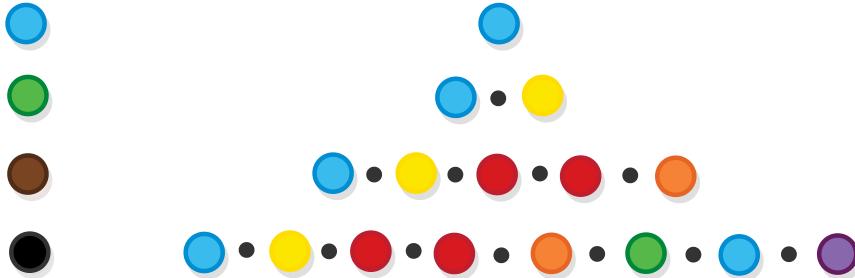
我们可以从一组彩色球中构造一个自由幺半群——该幺半群的元素将是所有可能的球的组合序列。



The free monoid with the set of balls as a generators

自由

幺半群是特殊的——在给定集合上的自由幺半群的每个元素都可以通过应用幺半群的规则,转换为任何其他使用相同生成元的幺半群的相应元素。例如,如果我们将上述元素应用于颜色混合幺半群的规则,得到如下结果。



Converting the elements of the free monoid to the elements of the color-mixing monoid

任务 (Task): 写出颜色混合幺半群的规则。

如果我们戴上程序员的帽子,会发现集合 T 上的自由幺半群类型(我们可以记作 `FreeMonoid<T>`)与类型 `List<T>`(如果你更喜欢的话,可以用 `Array<T>`)是同构的,并且我们上面描述的特殊性质的直觉实际上非常简单:将对象保存在列表中可以让你将它们转换为任何其他结构,即当我们想要对一堆对象进行某些操作,但我们不知道具体是什么操作时,我们只是将这些对象保存在列表中,直到执行操作的时机到来。

虽然自由幺半群的直觉似乎足够简单,但它的正式定义目前还不是我们可以处理的内容...因为我们还需要覆盖更多内容。

我们知道,作为给定生成集合中最一般的幺半群,自由幺半群可以转换为所有其他幺半群。即存在一个函数将自由幺半群映射到其他幺半群。但这个函数是什么呢?请继续关注接下来的章节。

{%endif%}

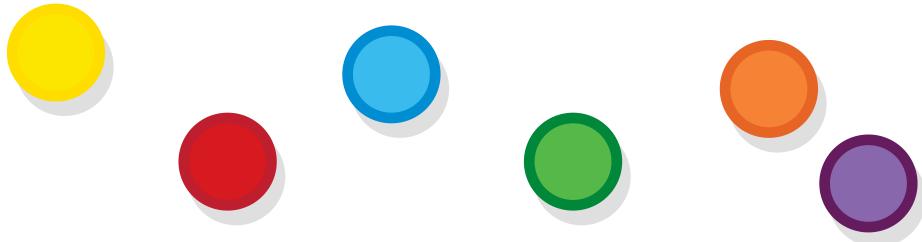
顺序 (Orders)

给定一组对象，可以基于许多不同的标准对它们进行排序（具体取决于对象本身）——如大小、重量、年龄、字母顺序等。

然而，目前我们并不关心用来对对象排序的标准，而是关心定义顺序的关系的本质，这种关系也有不同的类型。

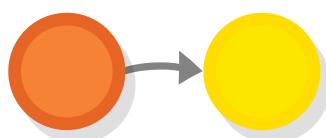
从数学上讲，顺序作为一个结构（类似于幺半群 (monoid)）由两个部分组成。

其中之一是一个事物的集合（例如彩色球），有时我们称之为顺序的底层集合。



Balls

另一个是这些事物之间的二元关系，通常用箭头表示。

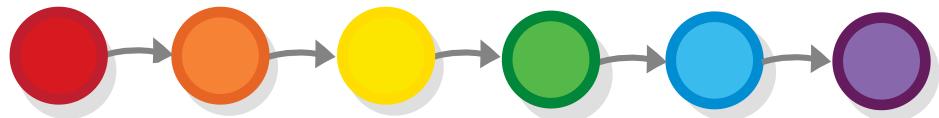


Binary relation

并不是所有的二元关系都是顺序——只有符合某些特定标准的关系才可以成为顺序。我们将在回顾不同类型的顺序时进行讨论。

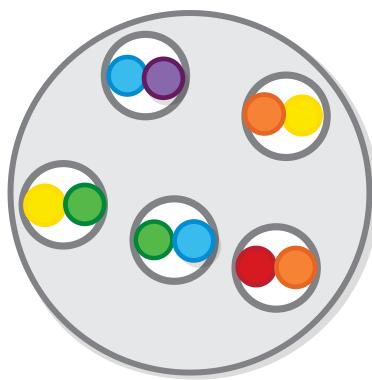
线性顺序 (Linear order)

让我们从一个例子开始——最直接的顺序类型是线性顺序, 即每个对象都有自己的位置, 并且可以与其他所有对象进行比较。在这种情况下, 排序标准是完全确定的, 不存在任何关于哪个元素在前哪个在后的模糊性。比如, 按颜色的光波长排序(或根据它们在彩虹中的顺序排序)。



Linear order

使用集合论 (set theory), 我们可以将这种顺序以及任何其他顺序表示为底层集合与其自身的笛卡尔积 (cartesian product)。



Binary relation as a product

在编程中, 顺序是通过提供一个函数来定义的, 该函数给定两个对象后, 告诉我们哪个对象“更大”(排在前面), 哪个“更小”。不难看出这个函数实际上是笛卡尔积的定义。

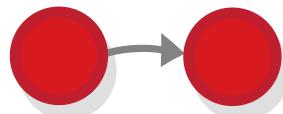
```
[1, 3, 2].sort((a, b) => {
  if (a > b) {
    return true
  } else {
    return false
  }
})
```

然而(这里开始变得有趣),并不是所有这样的函数(也不是所有笛卡尔积)都定义了顺序。为了真正定义一个顺序(例如,每次给出相同的输出,独立于最初对象的排列方式),函数必须遵守几个规则。

巧合的是(或者说这根本不是巧合),这些规则几乎等同于定义顺序关系标准的数学法则,即那些定义哪个元素可以指向哪个元素的规则。让我们来回顾一下它们。

自反性 (Reflexivity)

让我们先讨论最无趣的法则——每个对象都必须大于或等于它自己,或者说对于所有 $a, a \leq a$ (在公式中,元素之间的关系通常用 \leq 表示,但也可以用箭头表示,箭头从第一个对象指向第二个对象)。



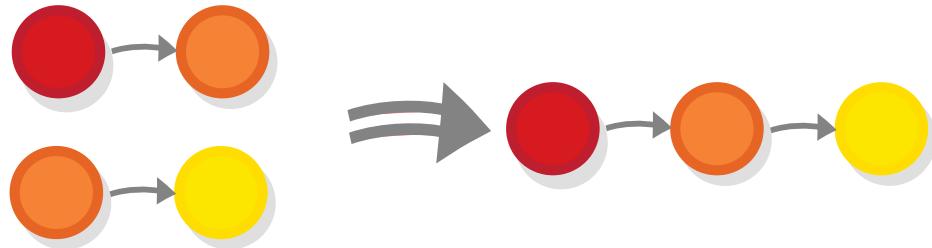
Reflexivity

这个法则存在的特殊理由并不多,除了“基本情况”应该以某种方式被涵盖。

我们也可以反过来表述这个法则,说每个对象不应该与自身具有这种关系,在这种情况下,我们会得到一个类似于大于而非大于或等于的关系,这样的顺序称为严格顺序。

传递性 (Transitivity)

第二个法则可能是最不明显的(但可能也是最关键的)——它指出, 如果对象 a 大于对象 b , 那么它也自动大于所有比 b 小的对象, 即 $a \leq b \wedge b \leq c \rightarrow a \leq c$ 。

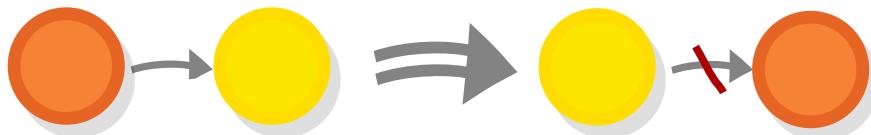


Transitivity

这是在很大程度上定义了什么是顺序的法则: 如果我踢足球比我祖母好, 那么我也应该比我祖母的朋友好, 否则我就不能说我踢得比她好。

反对称性 (Antisymmetry)

第三个法则称为反对称性 (antisymmetry)。它指出, 定义顺序的函数不应该给出矛盾的结果(换句话说, 如果 $x \leq y$ 且 $y \leq x$, 那么仅当 $x = y$ 时才成立)。



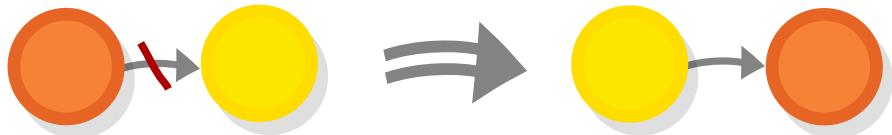
antisymmetry

这也意味着不允许平局——要么我踢足球比我祖母好，要么她踢得比我好。

完备性 (Totality)

最后一个法则称为完备性(或联通性(connexity))，它规定属于该顺序的所有元素必须是可比较的(即 $a \leq b \vee b \leq a$)。换句话说，对于任意两个元素，总有一个比另一个“更大”。

顺便说一句，这条法则使自反性法则成为多余的，因为自反性只是当 a 和 b 是同一个对象时完备性的特例，但我仍然想要讨论它，原因稍后会变得清晰。



connexity

实际上，原因是这样的：这条法则不像其他法则那样“板上钉钉”，即我们可能会想到某些不适用它的情况。例如，如果我们试图根据足球技能来对所有人排序，有很多方法可以对一个人相对于他们的朋友或他们朋友的朋友进行排名，但没有办法对从未互相踢过球的群体进行排序。

那些不遵循完备性法则的顺序称为偏序(partial orders)，(线性顺序也称为全序(total orders))。

问题： 我们之前讨论过一个与此非常相似的关系。你还记得它吗？有什么不同？

任务： 想一想你知道的一些顺序，并弄清楚它们是偏序还是全序。

偏序实际上比线性/全序更有趣。不过在我们深入讨论之前，先谈谈数字。

自然数的顺序 (The order of natural numbers)

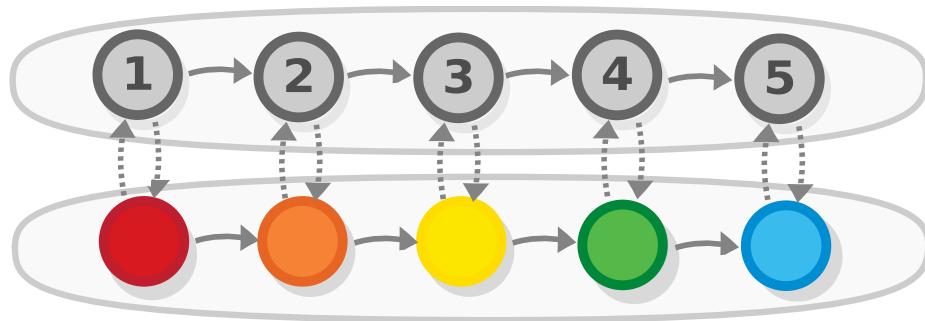
自然数在“大于或等于”运算下形成线性顺序（我们在公式中一直使用的符号）。



numbers

在许多方面，数字是典型的顺序——任何有限的对象顺序都是自然数顺序的同构 (isomorphic)，因为我们可以将任何顺序的第一个元素映射到数字 1，第二个元素映射到数字 2，等等（反向操作也是可以的）。

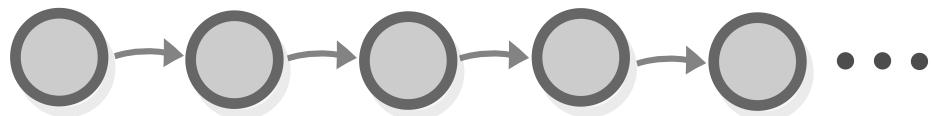
如果我们仔细思考，这种同构实际上比通过法则定义的顺序更接近日常所理解的线性顺序——当大多数人想到顺序时，他们并不是在想一个传递的、反对称的和完备的关系，而是想到一种基于某种标准决定哪个对象排在前面、哪个排在后面的顺序。因此，注意到这两者是等价的很重要。



Linear order isomorphisms

由于任何有限的对象顺序都与自然数同构，故所有相同大小的线性顺序都是相互同构的。

所以，线性顺序很简单，但从范畴论 (category theory) 的角度来看，这也是最无聊的顺序，特别是当我们看到所有有限的线性顺序（以及大多数无限顺序）都与自然数同构时，它们的图示看起来都一样。



Linear order (general)

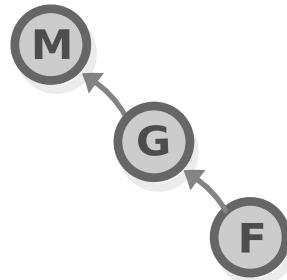
然而，偏序并非如此，我们将在接下来的内容中继续探讨。

偏序 (Partial order)

与线性顺序类似, 偏序由一个集合加上一个关系组成, 唯一的区别是, 尽管它依然遵循自反、传递和反对称的法则, 这个关系并不遵守完备性法则, 也就是说, 并非集合中的所有元素都一定是有序的。我说“不一定”是因为即使所有元素都是有序的, 它依然是一个偏序(就像群(group)依然是么半群(monoid)一样)——所有线性顺序都是偏序, 但反过来则不然。我们甚至可以创建一个顺序的顺序, 根据其一般性来排序。

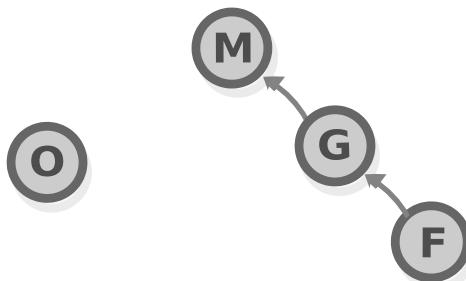
偏序还与我们在第一章中讨论的等价关系 (*equivalence relations*) 概念相关, 只不过对称性 (*symmetry*) 法则被反对称性 (*antisymmetry*) 所取代。

如果我们重温之前足球选手的排名列表的例子, 可以看到包含的只有我自己、我祖母和她的朋友时是一个线性顺序。



Linear soccer player order

然而, 加入一个我们谁也没有踢过比赛的另一位人选后, 层级变得非线性的, 即变成了偏序。

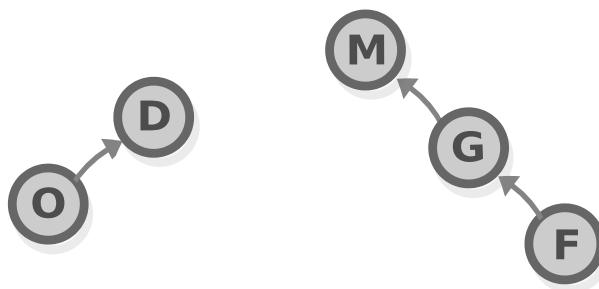


Soccer player order - leftover element

这就是偏序和全序的主要区别——偏序不能给出一个确定的答案，告诉我们谁比谁好。但有时候这正是我们需要的——在体育比赛和其他领域，并不总有一种适合线性评价的方式。

链 (Chains)

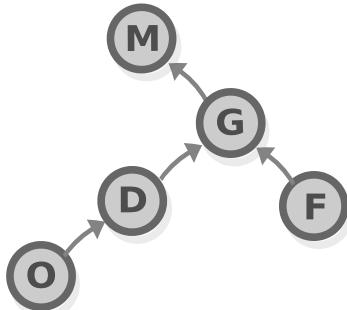
之前我们说过，所有线性顺序都可以用相同的链状图表示，我们可以反过来说，所有看起来不同于这种图的图都代表偏序。例如，偏序可能包含一些线性排列的子集，例如在我们的足球例子中，我们可以有不同的朋友群体，他们只在群体内部相互比赛并有各自的排名，但与其他群体没有排名关系。



Soccer order - two hierarchies

构成偏序的不同线性顺序被称为 **链**。在这个图中有两条链，分别是 $m \rightarrow g \rightarrow f$ 和 $d \rightarrow o$ 。

链不必完全彼此独立才能形成偏序。它们可以相互连接，只要这些连接并非完全一对一的，即最后一个元素从一条链连接到另一条链的第一个元素（这将有效地将它们统一为一条链）。



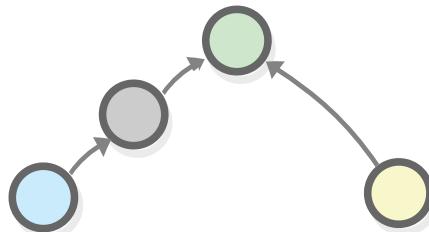
Soccer order - two hierarchies and a join

上面的集合不是线性排序的。因为尽管我们知道 $d \leq g$ 且 $f \leq g$, 但 d 和 f 之间的关系是未知的——它们中的任何一个都可能比另一个大。

最大和最小对象 (Greatest and least objects)

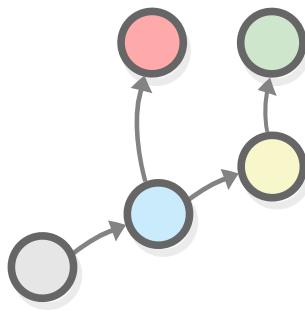
尽管偏序不能为我们提供关于“谁比谁好”的确定答案, 但其中一些仍然可以回答更为重要的问题(无论是在体育比赛还是其他领域), 即“谁是第一名?”也就是, 谁是冠军, 谁比其他人都好。或者更普遍地说, 哪个元素比其他所有元素都大。

我们称这样的元素为最大元素。一些(但不是所有)偏序确实有这样的元素——在我们上一个图中, m 是最大元素, 在下一个图中, 绿色元素是最大的。



Join diagram with one more element

有时我们有多个比所有其他元素都大的元素, 在这种情况下, 它们中没有一个是最大的。

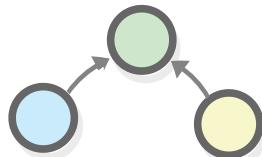


A diagram with no greatest element

除了最大元素之外, 偏序还可能有最小元素, 其定义方式与最大元素相同。

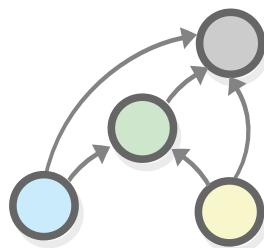
并 (Joins)

连接两个作为偏序一部分的元素的最小上界称为这两个元素的并(join), 例如, 绿色元素是另两个的并。



Join

可能有多个比 a 和 b 大的元素(所有比 c 大的元素也比 a 和 b 大), 但它们中只有一个并。形式上, a 和 b 的并定义为比 a 和 b 都大的最小元素(即对于 $a \leq c$ 且 $b \leq c$, c 是最小的)。

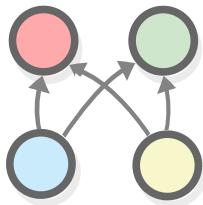


Join with other elements

给定任意两个元素，其中一个大于另一个（例如 $a \leq b$ ），并就是较大的那个元素（在这个例子中是 b ）。

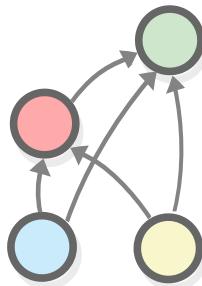
在线性顺序中，任意两个元素的并就是较大的那个元素。

和最大元素类似，如果两个元素有多个同样大的上界，那么它们中没有一个是并（并必须是唯一的）。



A non-join diagram

如果其中一个元素被确定为比其他元素小，它就立刻符合条件成为并。

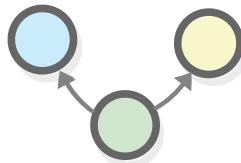


A join diagram

问题：范畴论中的哪个概念让你联想到并？

交 (Meets)

给定两个元素，比它们都小的最大元素称为这些元素的交 (meet)。



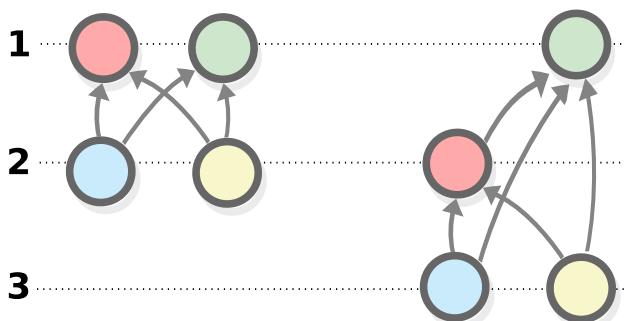
Meet

其规则与并的规则相同。

哈斯图 (Hasse diagrams)

我们在本节中使用的图称为“哈斯图” (Hasse diagrams)，它们的工作方式与我们通常的图类似，但有一个额外的规则——“较大”的元素总是位于较小元素的上方。

在箭头的层面上，这条规则意味着如果你向一个点添加箭头，箭头指向的点必须总是在发出箭头的点上方。



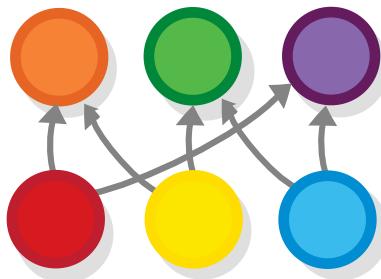
A join diagram

这种排列允许我们通过简单地查看哪个点在另一个点的上方来比较任意两个点。例如，我们可以通过识别它们连接到的元素并查看哪个元素最低，来确定两个元素的并。

颜色顺序 (Color order)

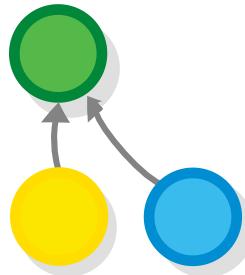
我们都知道很多全序的例子（任何形式的图表或排名都是全序），但我们很难想到一些显而易见的偏序例子。那么让我们来看一些例子。这将为我们提供一些背景，并帮助我们理解什么是并。

为了保持我们一贯的风格，让我们回顾一下颜色混合幺半群 (color-mixing monoid)，并创建一个颜色混合偏序，在其中所有颜色指向包含它们的颜色。



A color mixing poset

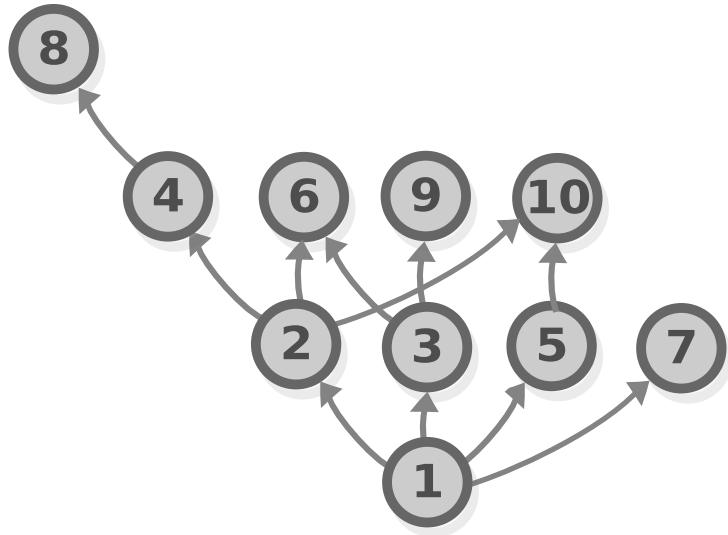
如果你仔细观察，会发现任意两个颜色的并就是它们混合后产生的颜色。很有趣，对吧？



Join in a color mixing poset

通过除法排序的数字 (Numbers by division)

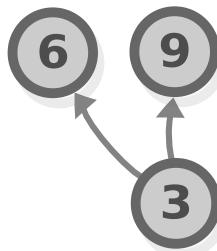
我们看到，当按“大于或等于”对数字排序时，它们形成线性顺序（甚至是线性顺序）。但数字也可以形成偏序，例如，如果我们根据哪个数字能整除哪个数字进行排序，那么它们就会形成一个偏序。例如，如果 a 能整除 b ，那么 a 就排在 b 前面。例如，因为 $2 \times 5 = 10$, 2 和 5 在 10 前面（但 3 不能在 10 前面）。



Divides poset

事实证明(实际上有充分的理由),在这个偏序中,并操作再次对应于与对象相关的操作——两个数字的并是它们的最小公倍数 (least common multiple)。

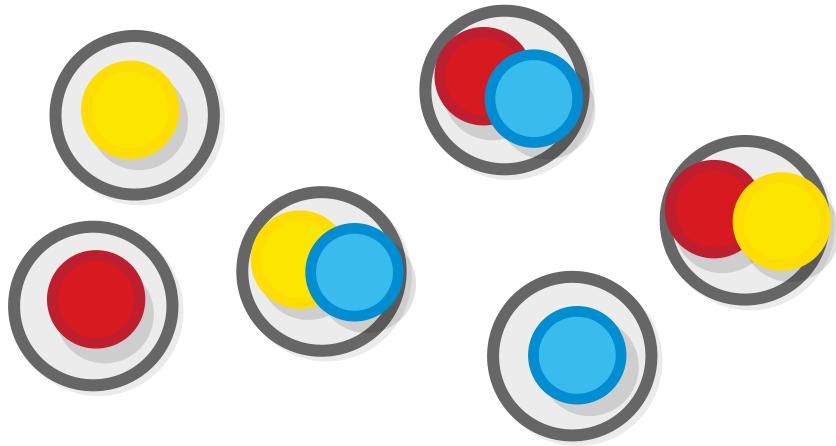
而两个数字的交(与并相反)是它们的最大公约数 (greatest common divisor)。



Divides poset

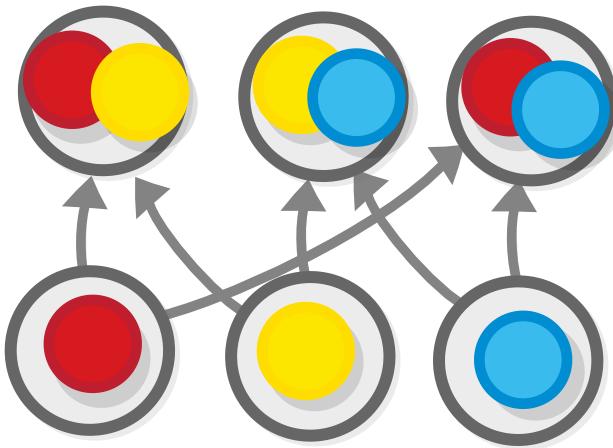
包含顺序 (Inclusion order)

给定包含给定元素组合的所有可能集合...



A color mixing poset, ordered by inclusion

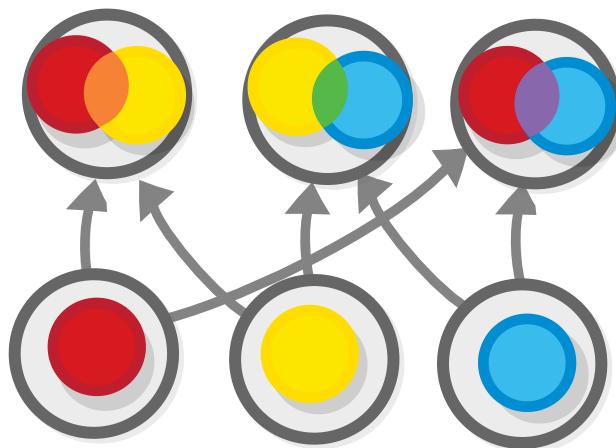
...我们可以定义这些集合的包含顺序(inclusion order), 其中如果 a 包含 b , 即 b 是 a 的子集, 那么 a 排在 b 之前。



A color mixing poset, ordered by inclusion

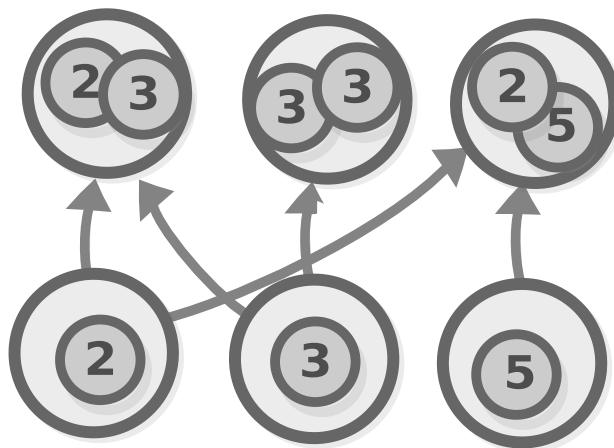
在这种情况下, 两个集合的并操作是它们的并集, 而交操作是它们的交集。

这个图可能会让你想起某些东西——如果我们将每个集合中包含的颜色混合成一种颜色, 我们就会得到前面看到的颜色混合偏序。



A color mixing poset, ordered by inclusion

使用数字除法的排序示例也与包含顺序同构, 具体来说是所有质数集合的包含顺序(包括重复的质数), 或者可以说是所有质数幂(prime powers)的集合。这一结论由算术基本定理(fundamental theorem of arithmetic)证实, 该定理指出, 每个数字都可以用质数的乘积以唯一的方式表示。

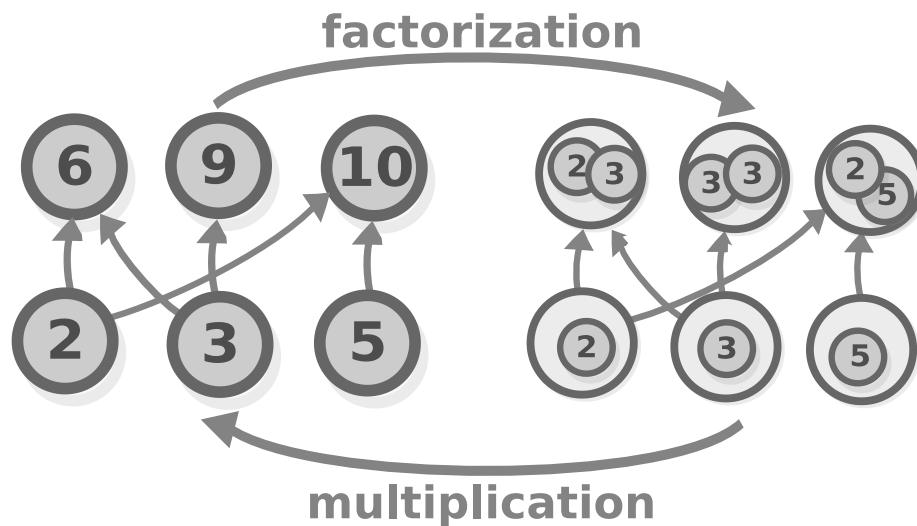


Divides poset

顺序同构 (Order isomorphisms)

我们已经多次提到顺序同构 (order isomorphisms), 所以是时候详细解释一下了。以数字偏序和质数包含顺序之间的同构为例。像任何两个集合之间的同构一样, 它由两个函数组成:

- 一个函数从质数包含顺序到数字顺序 (在这种情况下就是将集合中的所有元素相乘)。
- 一个函数从数字顺序到质数包含顺序 (即质因数分解, 这个操作找到一组质数, 这些质数相乘后得到该数字)。



Divides poset

顺序同构本质上是底层集合的同构 (可逆函数)。然而, 除了它们的底层集合, 顺序还包含连接它们的箭头, 所以还有一个额外的条件: 为了使一个可逆函数构成顺序同构, 它必须尊重这些箭头, 换句话说, 它应该保持顺序。更具体地说, 将这个函数 (称为 F) 应用于一个集合中的任意两个元素 (a 和 b) 时, 应该得到在另一个集合中具有相同对应顺序的两个元素 (即 $a \leq b$ 当且仅当 $F(a) \leq F(b)$)。

Birkhoff 表示定理 (Birkhoff's representation theorem)

到目前为止, 我们已经看到了两种不同的偏序, 一种基于颜色混合, 一种基于数字除法, 它们可以通过某些基本元素(第一种情况下是原色, 第二种情况下是质数或质数幂)的所有可能组合的包含顺序来表示。许多其他偏序也可以用这种方式定义。究竟是哪一些, Birkhoff 的一个惊人成果——*Birkhoff 表示定理*, 给出了答案。它们是符合以下两个条件的有限偏序:

1. 所有元素都有并和交。
2. 这些交和并运算彼此分配, 即如果我们用 \vee 或 \wedge 表示并和交, 那么 $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ 。

符合第一个条件的偏序称为格 (lattices)。符合第二个条件的称为分配格 (distributive lattices)。

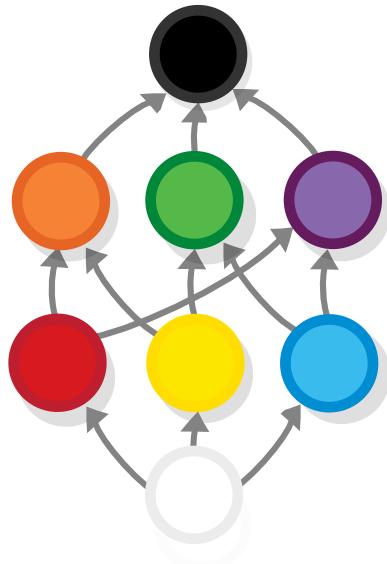
我们用来构造包含顺序的“质”元素是不能通过其他元素的并构造的元素。它们也被称为并不可约元素 (join-irreducible elements)。

顺便提一下, 不是分配格的偏序同样也与包含顺序同构, 只不过它们与不包含所有可能组合的元素的包含顺序同构。

格 (Lattices)

接下来我们将回顾 Birkhoff 定理适用的顺序, 即格。格是偏序的一种, 其中每两个元素都有并和交。因此, 每个格都是偏序, 但并非所有偏序都是格(我们还会看到更多这种层级结构的成员)。

大多数基于某种规则创建的偏序都是分配格, 例如前一节中的偏序就是当它们完整绘制时的分配格, 例如颜色混合顺序。



A color mixing lattice

注意, 我们在顶部添加了一个黑色球, 在底部添加了一个白色球。我们这样做是因为否则顶部三个元素将没有并元素, 而底部三个元素将没有交元素。

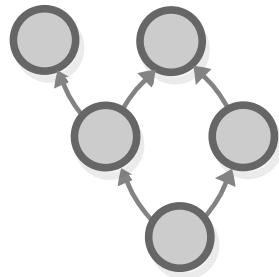
有界格 (Bounded lattices)

我们的颜色混合格有一个最大元素(黑色球)和一个最小元素(白色球)。拥有最小和最大元素的格称为有界格。不难看出, 所有有限格都是有界的。

任务：证明所有有限格都是有界的。

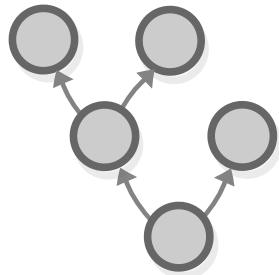
半格和树的插曲 (Interlude — Semilattices and Trees)

格是既有并又有交的偏序结构。只有并(没有交)或只有交(没有并)的偏序称为半格 (semilattices)。更具体地说，有交的偏序称为交半格 (meet-semilattices)。



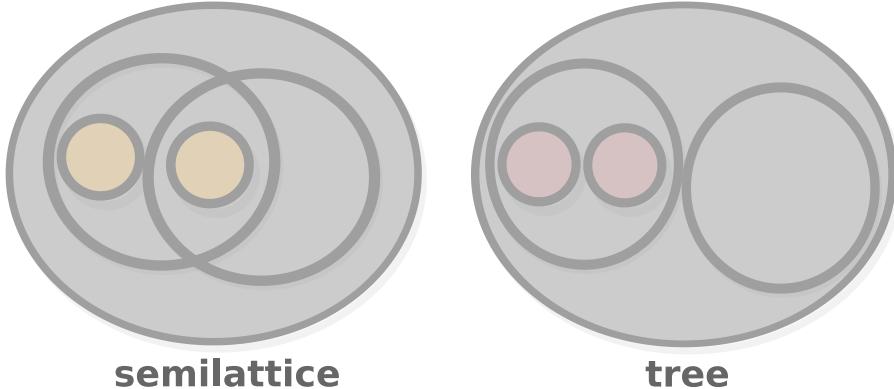
Semilattice

与半格类似 (而且可能比半格更为人熟知) 的结构是树 (tree)。



Tree

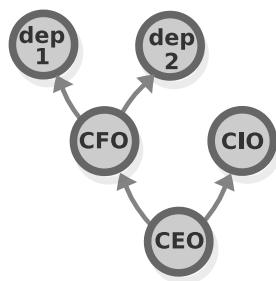
两者的区别很小，但至关重要：在树中，每个元素只能与一个其他元素相连（尽管它可以有多个元素连接到它）。如果我们将树表示为包含顺序，那么每个集合只会“属于”一个上位集合，而在半格中则没有这样的限制。



Tree and semilattice compared

一个直观的理解是，半格 (semilattice) 能够表示更为一般的关系。例如，母子关系形成了一棵树（一个母亲可以有多个孩子，但一个孩子只能有一个母亲），而“兄弟姐妹”关系则形成了一个格 (lattice)，因为一个孩子可以有多个哥哥或姐姐，反之亦然。

为什么我要讲树 (trees) 呢？因为人们倾向于用它们来建模各种现象，并将一切想象成树。树这种结构是我们每个人都理解的，它自然地展现在我们面前，甚至在我们还没有意识到自己在使用结构的时候就已经在使用它了——大多数人为设计的层级结构都是以树的形式进行建模的。一个典型的组织架构被建模为树形结构——你有一个人在顶端，有几个人向他汇报，然后更多的人向这几个人汇报。



Tree

（与非正式的社交群体相比，后者大致上每个人都与其他人都有联系。）

而且，城市（那些设计出来的，而不是自然演变出来的）也被建模为树形结构：你有几个街区，每个街区都有一所学校、一个商场等等，并且它们

彼此相连，在大城市中，它们又组织成更大的生活单元。

相对于使用格 (lattice) 来进行建模，使用树结构的倾向在Christopher Alexander的开创性文章《城市不是一棵树》中得到了详尽的探讨。

在结构的简洁性上，树类似于那种强迫症式的整洁与秩序感，这种秩序要求壁炉架上的烛台必须完全直立并对称地排列。与此相对，半格 (semilattice) 是复杂结构的代表；它是生命的结构，是伟大画作与交响乐的结构。

总的来说，似乎是那些由人类专门设计的层级结构（如城市）更倾向于以树的形式展现，而那些自然存在的层级结构（如颜色的层次结构）则更倾向于以格子的形式呈现。

插曲:形式概念分析 (Interlude: Formal concept analysis)

在上一节中, 我们(以及克里斯托弗·亚历山大)认为基于格 (lattice) 的层级结构是“自然的”, 也就是说它们在自然界中出现。现在我们将看到一种有趣的方法, 通过一组具有某些属性的对象来揭示这种层级结构。这是一个称为形式概念分析 (formal concept analysis) 的数学方法概述。

我们将要分析的数据结构称为形式背景 (formal context), 它由 3 个集合组成。首先是包含我们将要分析的所有对象的集合 (记为 G)。



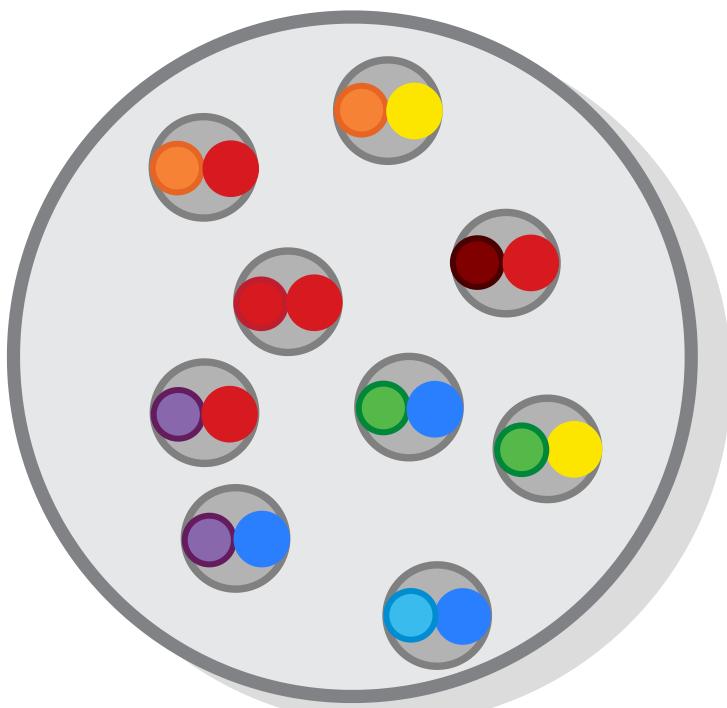
Formal concept analysis - function

其次是一组这些对象可能具有的属性 (attributes) (记为 M)。在这里, 我们将使用三种基本颜色。



Formal concept analysis - function

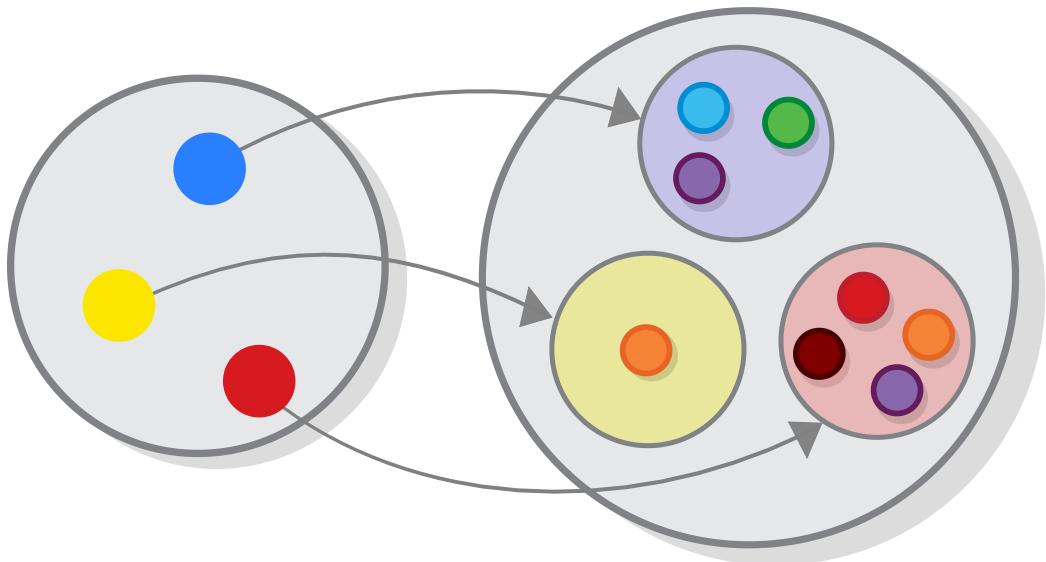
最后是一个关系集 (relation set) (称为关联 (incidence)), 它表示哪些对象具有哪些属性, 并通过一组对 $G \times M$ 表示。因此, 包含一个球和一种基本颜色 (例如黄色) 的对表示该球的颜色包含黄色 (即由黄色及其他颜色组成)。



Formal concept analysis - function

现在，让我们使用这些集合来构建一个格。

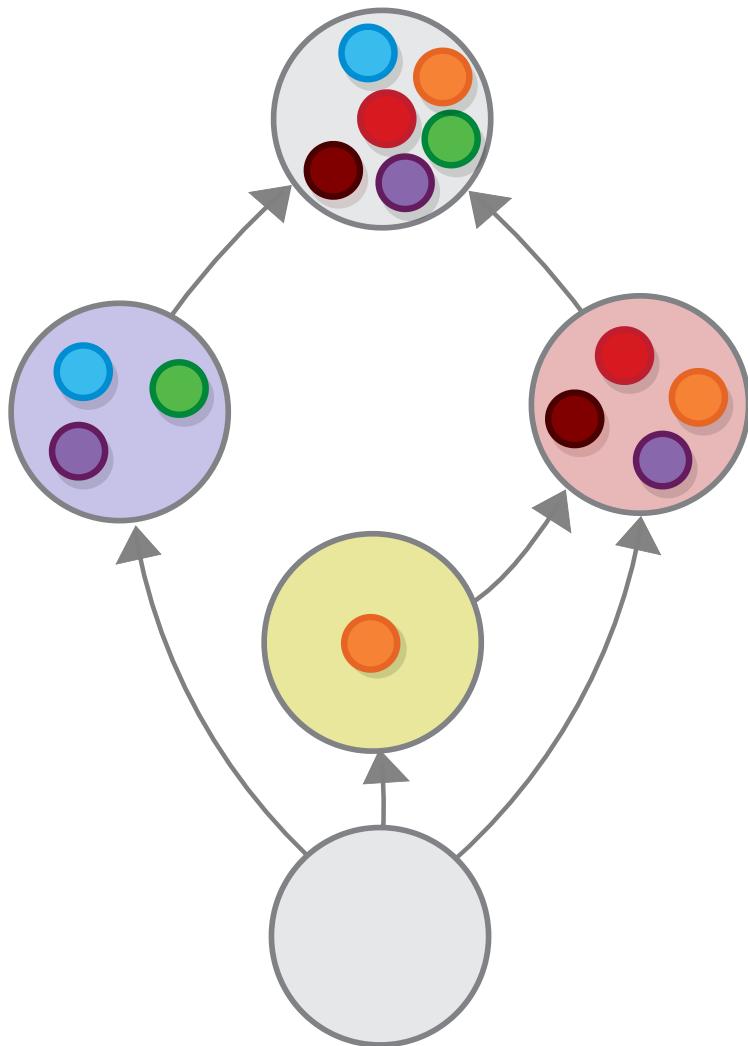
第一步：从这一对集合中，我们构建一个函数，将每个属性与共享该属性的对象集合相关联（我们可以这样做，因为函数是关系，而关系是由对表示的）。



Formal concept analysis - function

看一下这个函数的目标，即一组集合。有没有什么方法可以对这些集合进行排序以便更好地可视化？当然，我们可以通过包含关系来排序这些集合。这样，每个属性都会与一个由相似对象组成的属性相连。

加上顶点和底点，我们得到了一个格。



Formal concept analysis - function

将概念按格的方式排序可能有助于我们在上下文中看到概念之间的联系。例如，我们可以看到我们集合中所有包含黄色的球也包含红色。

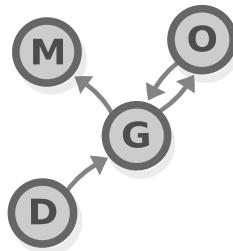
任务：选择一组对象和一个包含属性的集合，创建你自己的概念格。
例如：对象可以是生命体：鱼、青蛙、狗、水草、玉米等，属性可以是它们的特征：“生活在水中”、“生活在陆地上”、“可以移动”、“是植物”、“是动物”等等。

预序 (Preorder)

在上一节中, 我们看到从(线性)顺序的规则中删除全体性法则会产生一种不同(且更有趣)的结构, 称为偏序(partial order)。现在让我们看看, 如果我们删除另一个法则, 即反对称性(antisymmetry)法则, 会发什么。如果你还记得, 反对称性法则规定, 不能同时有一个对象既小于又大于另一个对象(即 $a \leq b \iff b \not\leq a$)。

线性顺序 (Linear order)	偏序 (Partial order)	预序 (Preorder)
反身性 (Reflexivity) X	X	X
传递性 (Transitivity) X	X	X
反对称性 (Antisymmetry) X	X	
全体性 (Totality) X		

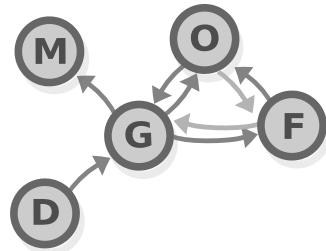
结果就是产生了一种称为预序的结构, 这并不是我们日常意义上的顺序——它可以从任何点到达任何其他点。如果偏序可以用来建模谁在足球比赛中比谁更强, 那么预序可以用来建模谁击败了谁, 无论是直接(通过比赛)还是间接。



preorder

预序只有一个规则——传递性 $a \leq b \wedge b \leq c \rightarrow a \leq c$ (如果我们算上反身性的话, 则有两个规则)。关于间接胜利的部分正是这一规则的结果。由于这个规则, 所有间接胜利(即不是直接对抗, 而是通过击败对手的

对手而获胜)都会作为其应用的直接结果自动添加,如下所示(我们用较浅的色调显示间接胜利)。



preorder in sport

因此,所有“循环”关系(例如较弱的玩家击败较强的玩家)最终都导致一组彼此互相连接的对象。

这一切结构都自然地源自简单的传递性法则。

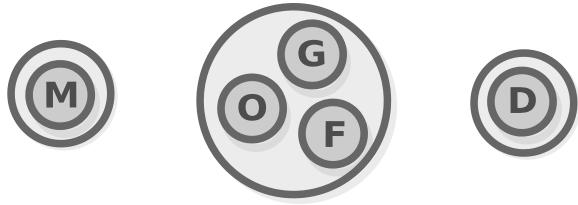
预序与等价关系 (Preorders and equivalence relations)

预序可以被看作是偏序 (partial orders) 和等价关系 (equivalence relations) 之间的中间地带。它们缺少的正是这两种结构的差异之处——即(反)对称性 (antisymmetry / symmetry)。因此,如果在预序中有一组对象满足对称性法则,那么这些对象就形成了一个等价关系。而如果它们满足反对称性法则,它们则形成了一个偏序。

等价关系 (Equivalence relation)	预序 (Preorder)	偏序 (Partial order)
反身性 (Reflexivity)	反身性 (Reflexivity)	反身性 (Reflexivity)
传递性 (Transitivity)	传递性 (Transitivity)	传递性 (Transitivity)
对称性 (Symmetry)	-	反 对 称 性 (Antisymmetry)

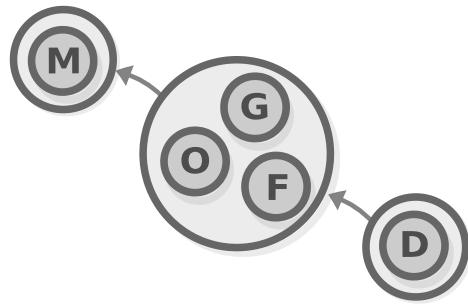
特别是,任何同时双向连接的一组对象(如上例所示)都遵循对称性要求。因此,如果我们将所有具有这种连接的元素进行分组,我们将得到一

组定义了基于预序的不同等价类 (equivalence classes) 的集合。



preorder

更有趣的是, 如果我们将预序中各个集合间的连接转化为这些集合之间的连接, 这些连接将遵循反对称性要求, 从而形成一个偏序。

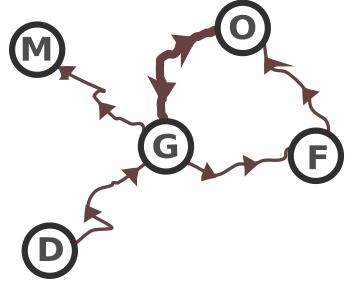


preorder

简而言之, 对于每一个预序, 我们都可以定义出该预序的等价类的偏序。

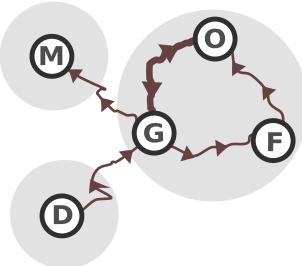
地图作为预序 (Maps as preorders)

我们经常使用地图进行导航, 往往不会意识到它们实际上是图 (diagrams)。更具体地说, 有些地图是预序——对象表示城市或交叉口, 关系则表示道路。



A map as a preorder

反身性反映了这样的事实:如果你有一条路可以从点 a 到点 b , 又有一条路可以从 b 到 c , 那么你也可以从 a 到 c 。双向道路可以用两条箭头表示, 它们在对象之间形成同构。那些无论如何都可以相互到达的对象组成了等价类(理想情况下, 所有交叉口都应该属于一个等价类, 否则就会出现无法返回的地方)。

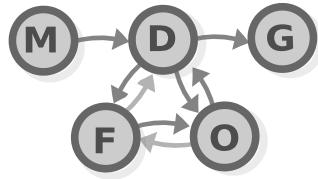


preorder

然而, 包含不止一条连接两个交叉口的路的地图(甚至包含不止一条路线)无法用预序表示。为此, 我们需要范畴(categories)(别担心, 我们会在后面讲到)。

状态机作为预序 (State machines as preorders)

现在, 让我们将前两个例子中的预序重新格式化为从左到右的哈斯图(Hasse diagram)。现在, 它(希望看起来)不再像一个层级结构, 也不像一张地图, 而像一个过程的描述(如果你细想一下, 过程实际上也是一种地图, 只不过是时间上的而不是空间上的)。这是描述一种称为有限状态机(finite state machine)的计算模型的非常好的方式。



A state machine as a preorder

有限状态机的规格由机器可以具有的一组状态组成,正如其名称所示,这些状态必须是有限的,还有一组过渡函数 (transition functions),指定我们会转移到哪个状态(通常以表格形式表示)。

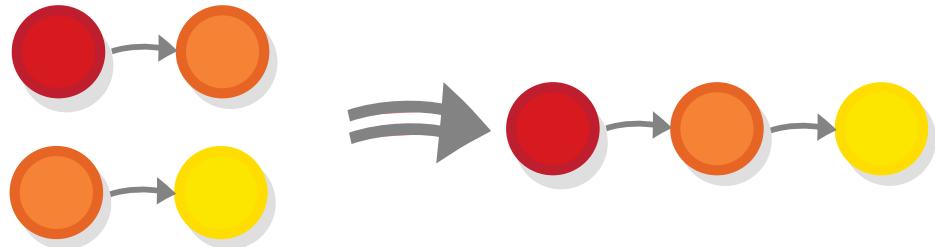
但正如我们所看到的,有限状态机类似于一个具有最大和最小对象的预序,在其中对象之间的关系由函数表示。

有限状态机在组织规划中广泛使用,例如,设想一个流程:某个产品被制造出来后,由质检员检查,如果他们发现某些缺陷,则将其交给修理部门,修理后再次检查,然后发送到发货部门。这个流程可以用上面的图来建模。

任务: 创建你自己的有限状态机,并展示如何将其视为预序的一部分。

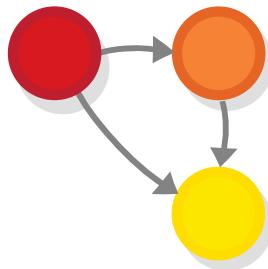
顺序作为范畴 (Orders as categories)

我们已经看到了预序是一个强大的概念，现在让我们深入研究一下控制它们的法则——传递性法则。这个法则告诉我们，如果我们有两对关系 $a \leq b$ 和 $b \leq c$ ，那么我们就自动拥有了第三对关系 $a \leq c$ 。



Transitivity

换句话说，传递性法则告诉我们， \leq 关系是可以复合的。也就是说，如果我们将“比……大”的关系视为态射 (morphism)，我们会发现传递性法则实际上就是范畴论中复合 (composition) 的定义。



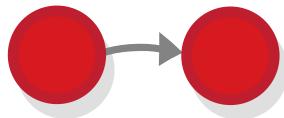
Transitivity as functional composition

(我们还必须验证该关系是结合的，但这很容易)

现在，让我们再次回顾范畴的定义。

一个范畴是由对象 (objects) (我们可以将其视为点) 和态射 (morphisms) (箭头) 组成的, 它们从一个对象指向另一个对象, 其中: 1. 每个对象必须有一个恒等态射 (identity morphism)。2. 必须有一种方法可以将两个具有适当类型签名的态射组合成一个第三个态射, 并且这种组合是结合的。

看起来我们已经满足了第二条法则。至于另一条——恒等性法则呢? 我们同样也拥有它, 以反身性的形式。



Reflexivity

因此, 这很明显——预序是范畴。这一点听起来也很自然, 尤其是在我们看到顺序可以通过包含顺序简化为集合与函数, 而集合与函数本身也构成了一个范畴。

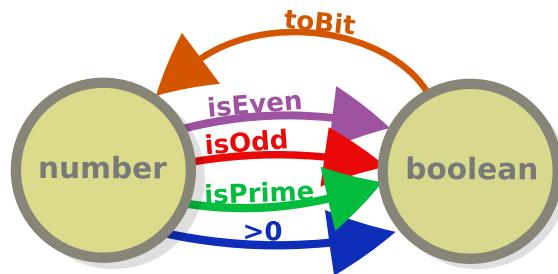
由于偏序 (partial orders) 和全序 (total orders) 也是预序, 它们同样是范畴。

当我们比较顺序的范畴与其他范畴时 (例如集合的范畴 (category of sets)), 我们会立即看到一个明显的区别: 在其他范畴中, 两个对象之间可能存在多个不同的态射 (箭头), 而在顺序中, 两个对象之间最多只能有一个态射, 也就是说, 我们要么有 $a \leq b$, 要么没有。



Orders compared to other categories

与之相对，在集合范畴中，可能存在无限多的函数，从比如整数的集合到布尔值的集合，并且存在从另一个方向映射回来的许多函数，而且这两个集合之间的函数是否存在并不意味着其中一个集合“比”另一个更大。



Orders compared to other categories

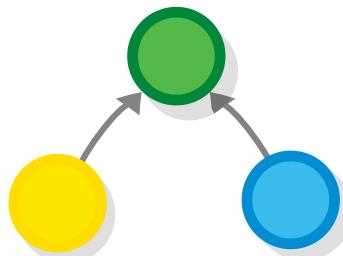
因此，顺序是一个在两个对象之间最多有一个态射的范畴。但反过来也是如此——每一个在两个对象之间最多只有一个态射的范畴也是一个顺序（在范畴论术语中，称为稀薄范畴（thin categories））。

由于在稀薄范畴中两个对象之间最多只有一个态射，因此一个有趣的推论是，所有的图表 (diagrams) 都是交换的。

任务：证明这一点。

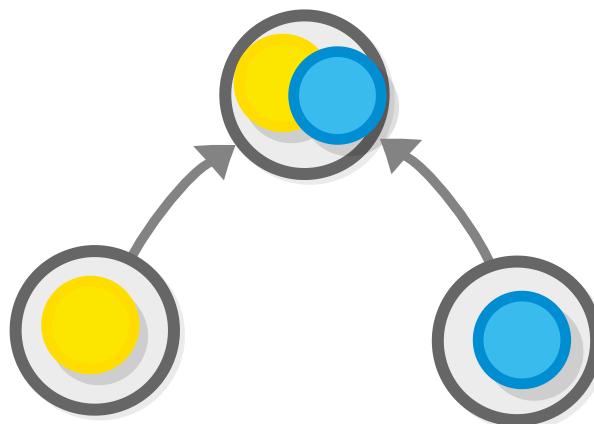
积与余积 (Products and coproducts)

当我们回顾前几章的图表时，让我们来看看第二章中定义余积 (coproduct) 的图表。



Joins as coproduct

如果你还记得,这是一种操作,对应于集合范畴中的包含(set inclusion)。

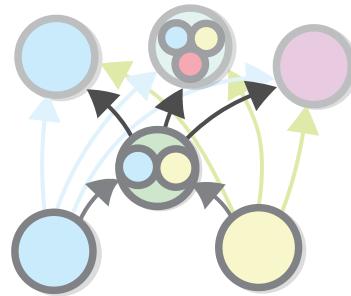


Joins as coproduct

但等等,这不是有另一种操作也对应于集合的包含吗?哦,对了,是顺序中的并(join)操作。而且,不仅如此,顺序中的并操作定义方式与范畴中的余积完全相同。

在范畴论中,物体 G 是物体 Y 和 B 的余积,如果满足以下两个条件:

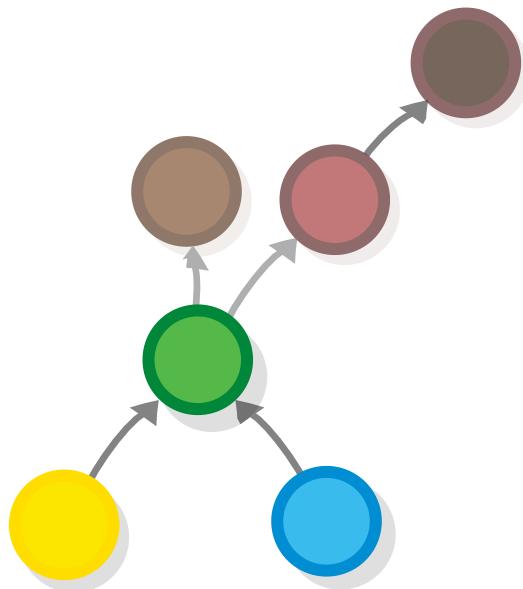
1. 我们有从每个元素到余积的态射,因此 $Y \rightarrow G$ 和 $B \rightarrow G$ 。
2. 对于任何另一个物体 P ,如果该物体也有这些态射(即 $Y \rightarrow P$ 和 $B \rightarrow P$),我们就有态射 $G \rightarrow P$ 。



Joins as coproduct

在顺序领域中,我们说物体 G 是物体 Y 和 B 的并,如果:

1. 它大于这两个物体，因此 $Y \leq G$ 和 $B \leq G$ 。
2. 它小于所有大于它们的物体，即对于任何其他物体 P ，如果 $P \leq G$ 且 $P \leq B$ ，则我们也有 $G \leq P$ 。



Joins as coproduct

我们可以看到这两个定义及其图表是相同的。因此，用范畴论的术语来说，我们可以说顺序中的范畴余积就是并操作。这当然意味着积(products) 对应于交(meets)。

总的来说，顺序(稀薄范畴)经常用于在更容易理解的上下文中探索范畴概念。例如，理解顺序理论中的并与交的概念，可以帮助你更好地理解更一般的范畴论中的积与余积概念。

当顺序被用作稀薄范畴时，它们同样是有用的，特别是在我们对从一个对象到另一个对象的态射之间的区别不太感兴趣的上下文中。在下一章中，我们将看到一个例子。

逻辑 (Logic)

现在让我们讨论另一个看似无关的话题，这样我们可以在意识到它实际上是在范畴论时“惊喜”自己。顺便说一下，本章除了这个“惊喜”之外，还会有另一个惊喜，所以不要打瞌睡。

此外，我不仅会带你进入数学的另一个分支，还会带你进入一个完全不同的学科——逻辑 (logic)。

什么是逻辑 (What is logic)

逻辑是关于可能性的科学。正因为如此，它是所有其他科学的基础，而其他科学则是关于现实的科学，即那些真实存在的事物。例如，科学解释了我们的宇宙是如何运作的，而逻辑则是描述中适用于任何其他可能存在宇宙的部分。科学理论的目标是与自身及观察结果保持一致，而逻辑理论只需与自身一致即可。

逻辑研究的是那些通过已知一件事情，推断出（或证明）另一件事情也为真的规则，而这些规则无关事物的领域（例如科学学科），只涉及它们的形式。

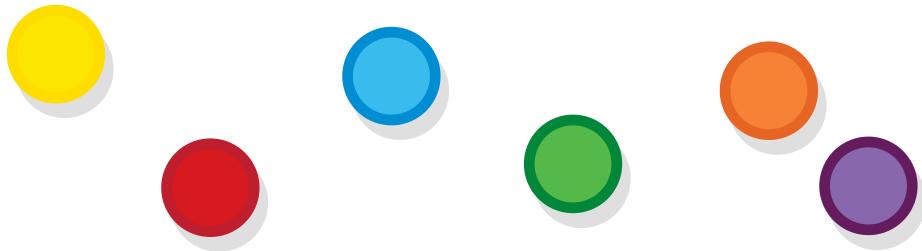
此外，逻辑试图将这些规则组织成逻辑系统（或称为形式系统）。

逻辑与数学 (Logic and mathematics)

看到这个描述，我们可能会认为逻辑的主题与我们在第一章中描述的集合论 (set theory) 和范畴论 (category theory) 非常相似——我们没有使用“形式”这个词，而是使用了另一个类似的词，即“抽象”，而“逻辑系统”的说法与“理论”类似。这个观察是非常正确的——今天，大多数人都同意每一个数学理论实际上就是逻辑，加上一些额外定义。例如，集合论 (set theory) 作为数学基础理论之一的部分原因是，它可以通过在标准逻辑公理中只添加一个原始定义——表示集合隶属 (set membership) 的二元关系，来定义。范畴论与逻辑也有密切关系，但它们的方式完全不同。

基本命题 (Primary propositions)

逻辑作为可能性的科学的一个后果是，要想在逻辑中有所作为，我们需要先有一组我们接受为真或假的命题。这些命题也被称为“前提” (premises)、“基本命题” (primary propositions) 或“原子命题” (atomic propositions)，如维特根斯坦所称。



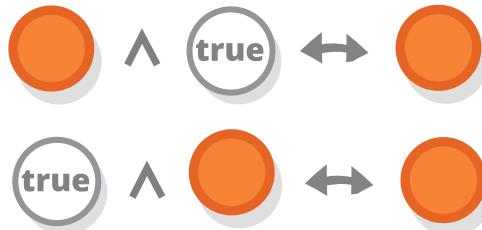
Balls

在逻辑的背景下,这些命题是抽象的(即我们并不直接关心它们的内容),因此可以用你熟悉的彩色球来表示。

组合命题 (Composing propositions)

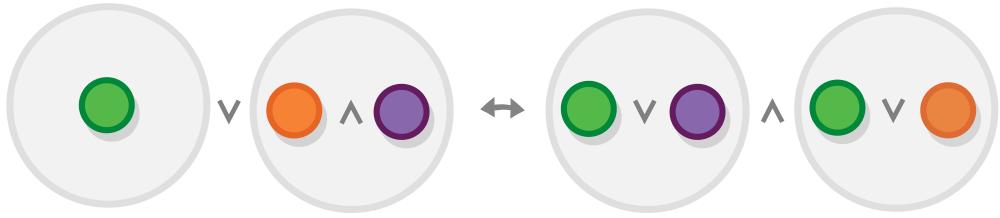
逻辑的核心,正如范畴论一样,是组合 (composition) 的概念——如果我们有两个或多个彼此相关的命题,我们可以通过逻辑运算符将它们组合成一个新的命题,例如“与” (and)、“或” (or)、“推导” (follows) 等。这样就得到了新的命题,我们可以称其为复合命题 (composite propositions) 以强调它们并非基本命题。

这种组合类似于两个幺半群对象通过幺半群操作组合成一个新对象的方式。事实上,一些逻辑运算确实构成幺半群,例如“与”运算,其中 *true* 命题作为幺元。



Logical operations that form monoids

然而,不同于幺半群/群,逻辑研究的不仅仅是一个运算符的组合,还研究多个逻辑运算符之间的相互关系。例如,在逻辑中,我们可能会关注“与”和“或”运算的分配律及其带来的影响。



The distributivity operation of “and” and “or”

需要注意的是, \wedge 是表示与 (and) 的符号, 而 \vee 是表示或 (or) 的符号 (尽管即使交换“与”和“或”, 上述规律仍然成立)。

基本命题与复合命题的等价性 (The equivalence of primary and composite propositions)

在观察上一个图时, 重要的是要强调, 尽管复合命题是由多个前提组成的(用包含其他小球的灰色小球表示), 它们并不与“基本命题”有任何本质区别(用单色小球表示), 并且它们的组合方式是相同的(尽管在最左边的命题中, 绿色小球包裹在灰色小球中以使图看起来更漂亮)。

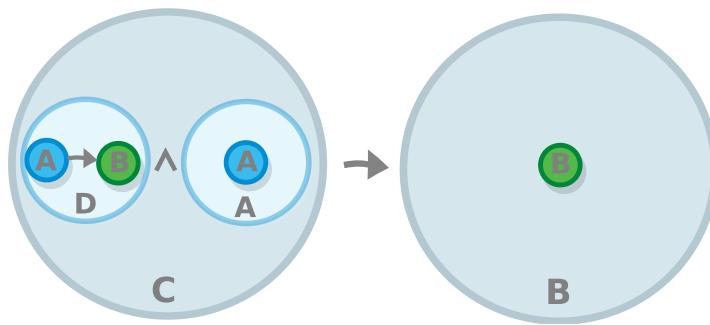


Balls as propositions

推理规则 (Modus ponens)

作为包含多个层次嵌套的命题的一个例子(并且也是逻辑学中非常著名的命题), 我们来看看推理规则(modus ponens)。推理规则是由两个命题组成的(我们将其标记为 A 和 B), 其表述为: 如果命题 A 为真, 并且命题 $(A \rightarrow B)$ 为真(即 A 推导出 B), 那么 B 也为真。例如, 如果我们知

道“苏格拉底是人”，并且知道“人都是会死的”（即“是人意味着会死”），我们就可以知道“苏格拉底会死”。



Modus ponens

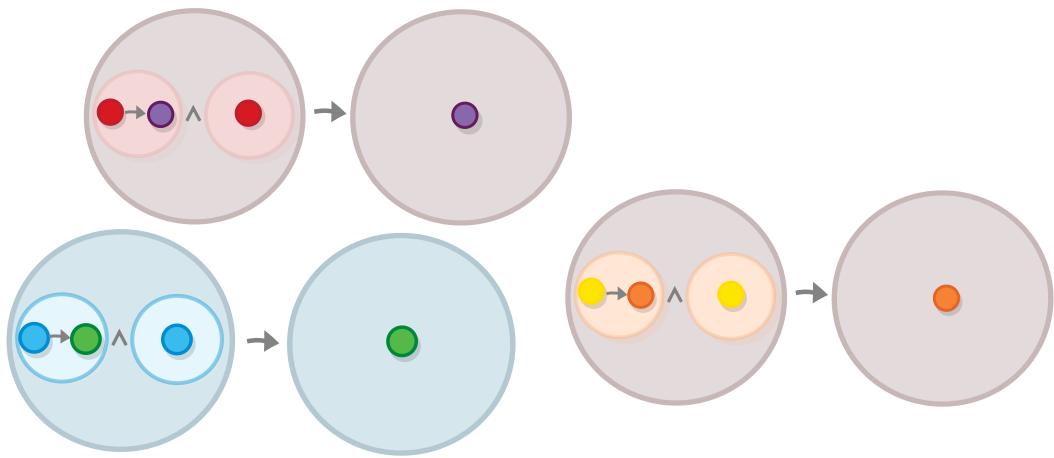
我们可以深入研究这个命题。我们可以看到它由两个命题通过“推导”(follows)关系组成，其中后继的命题(B)是基本命题，而其前导命题不是基本命题(我们称之为 C)，因此整个命题为 $C \rightarrow B$ 。

如果再往下深入一层，我们会发现 C 命题本身由两个命题通过“与”关系组成—— A 和另一个命题 D (因此 $A \wedge D$)，而 D 本身又由两个命题通过“推导”关系组成—— $A \rightarrow B$ 。但这一切用图示表现更为清晰。

必然真理 (Tautologies)

通常，我们无法在不知道一个复合命题所包含命题的真假之前，判断该复合命题的真假。然而，对于像推理规则这样的命题，我们可以：推理规则始终为真，无论组成它的命题是真还是假。如果我们想更正式一点，可以说它在逻辑系统的所有模型中都为真，模型是由我们的命题所代表的现实前提的集合。

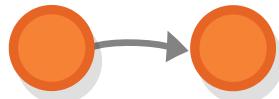
例如，之前的例子，即使我们将“苏格拉底”换成任何其他名字，或者将“必死”替换为人类拥有的其他任何特性，这个推理仍然成立。



Variation of modus ponens

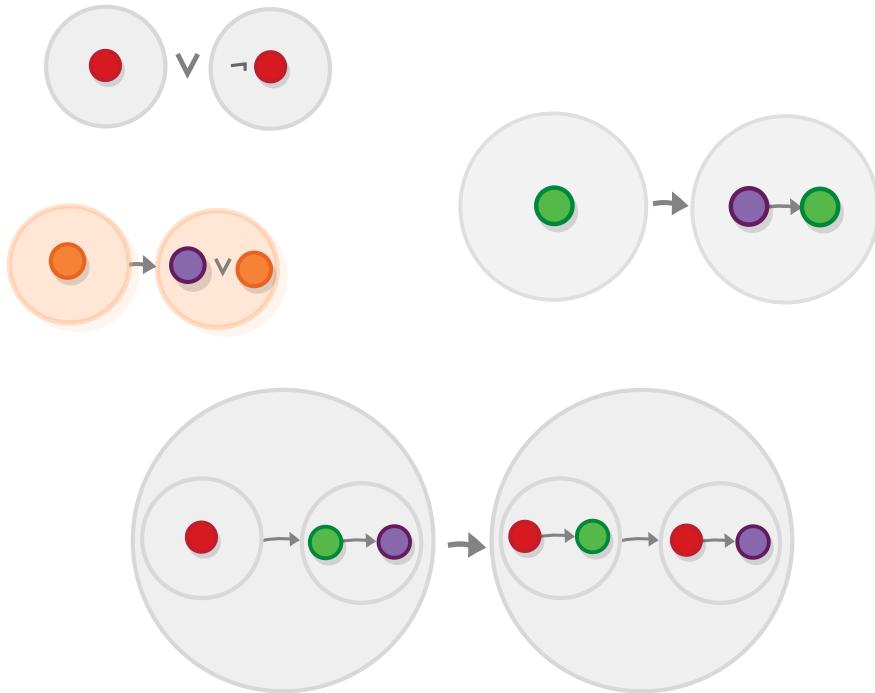
总是真命题被称为必然真理 (tautologies), 而那些永远为假的命题则被称为矛盾命题 (contradictions)。通过添加一个“非” (not), 你可以将必然真理变为矛盾命题, 反之亦然。

最简单的必然真理是所谓的同一律, 即每个命题都蕴含自身 (例如, “所有单身汉都是未婚的”)。这可能让你想起了一些东西。



Identity tautology

以下是一些更复杂 (不那么无聊) 的必然真理 (符号 \neg 表示“非”/否定) :



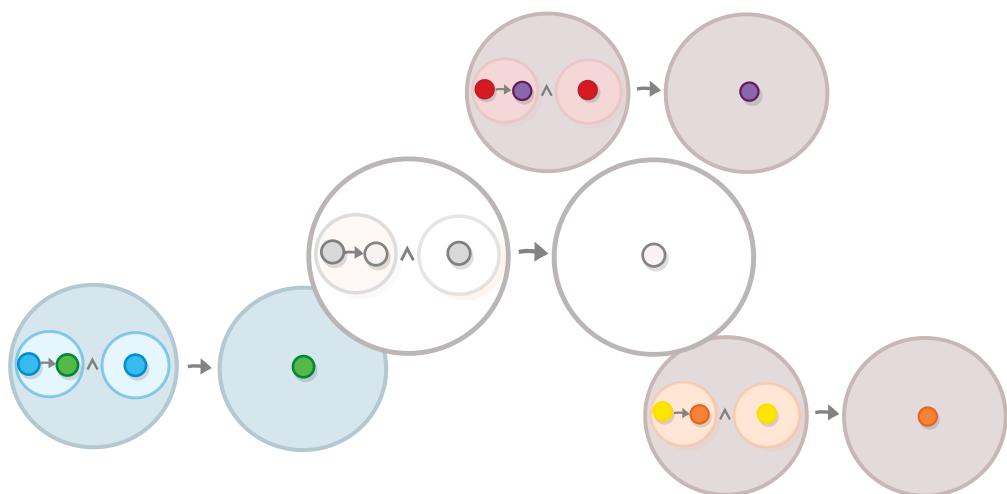
Tautologies

我们稍后将学习如何确定哪些命题是必然真理，但首先让我们看看为什么这些是重要的——即必然真理有什么用处。

公理模式/推理规则 (Axiom schemas/Rules of inference)

必然真理之所以有用，是因为它们是公理模式 (axiom schemas)/ 推理规则 (rules of inference) 的基础。公理模式或推理规则作为出发点，允许我们通过替换生成其他真实的逻辑命题。

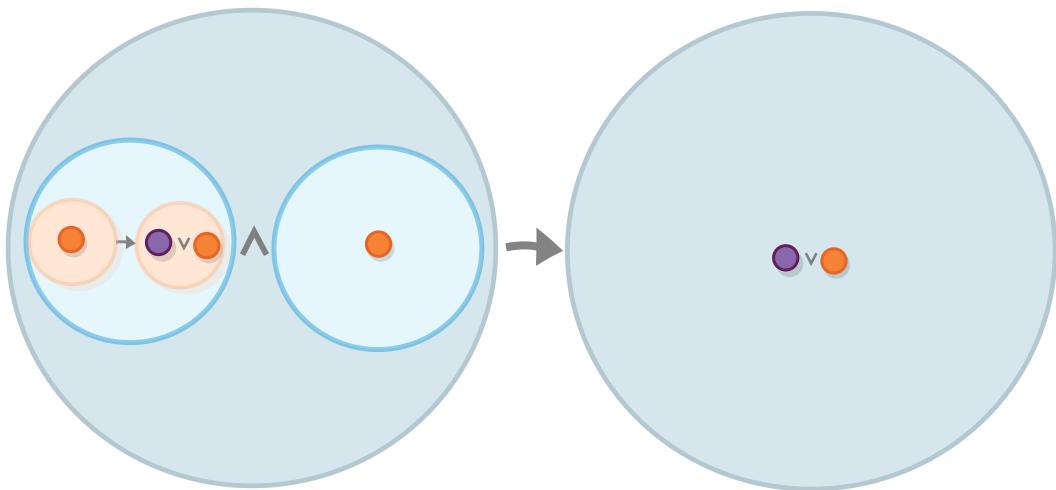
意识到推理规则中的小球颜色只是表面上的，我们可能希望表示出推理规则中所有变化共享的通用结构。



Modus ponens

这种结构（在我们的例子中类似于涂色书）被称为公理模式 (axiom schema)。由此模式生成的命题称为公理 (axioms)。

需要注意的是，我们插入模式中的命题不必是基本命题。例如，给定命题 a (用下图的橙色小球表示) 和命题 $a \rightarrow a \vee b$ (这是我们之前看到的必然真理之一)，我们可以将这些命题插入推理规则中，并证明 $a \vee b$ 为真。



Using modus ponens for rule of inference

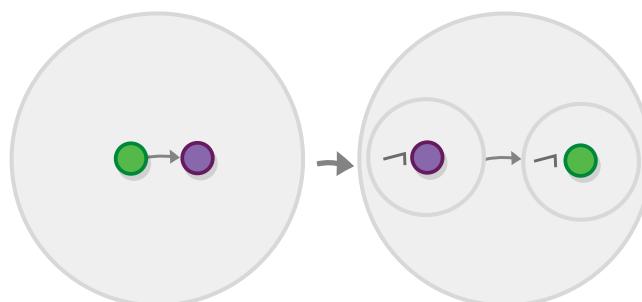
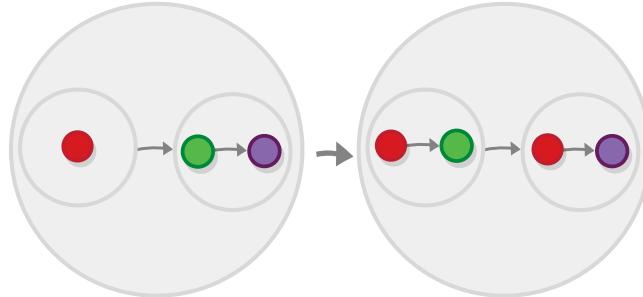
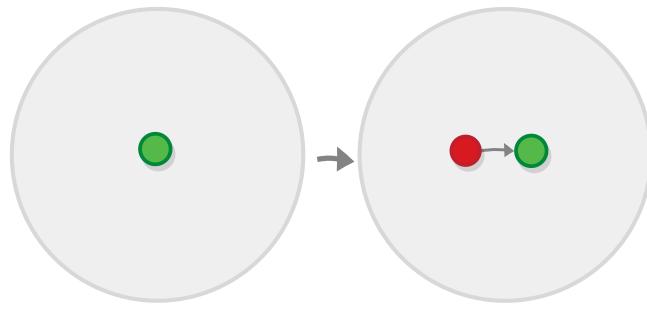
公理模式和推理规则几乎是相同的，唯一的区别在于推理规则允许我们从前提中实际提炼出结论。例如在上述情况下，我们可以使用推理规则证明 $a \vee b$ 为真。

所有公理模式都可以很容易地应用为推理规则, 反之亦然。

逻辑系统 (Logical systems)

既然我们知道可以使用公理模式/推理规则生成新的命题, 我们可能会问是否可以创建一个小的公理模式/推理规则集合, 这个集合经过精心策划, 能够生成所有其他命题。你可能会高兴(虽然也可能有点烦)地知道, 不仅仅存在一个这样的集合, 而是存在多个这样的集合。是的, 这类集合我们称之为逻辑系统(logical systems)。

以下是一个这样的集合, 它由以下五个公理模式以及推理规则推理规则组成(尽管我们用了颜色, 这些都是公理模式)。



A minimal collection of Hilbert axioms

证明该逻辑系统和其他类似系统是完备的（即能够生成所有其他命题）是哥德尔的功劳，这被称为“哥德尔完备性定理”(Gödel's completeness theorem)（哥德尔如此重要，我特意找到了“ö”这个字母，以确保拼写正确）。

结论 (Conclusion)

我们现在对一些主要的逻辑构造（公理、推理规则）的工作原理有了一个初步的认识。但为了证明它们为真，并理解它们是什么，我们需要通过特定的解释来进行。

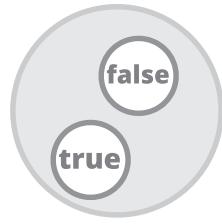
我们将研究两种解释——一种非常古老，另一种则相对较新。这将是一次稍微偏离我们通常的点与箭头主题的探讨，但我保证，这会是值得的。那么，让我们开始吧。

经典逻辑：真值功能解释 (Classical logic. The truth-functional interpretation)

在我们日常生活中感知的世界之外，存在着一个形式世界 (world of forms)，在那个世界中居住着所有的观念和概念，这些观念和概念体现在我们感知的物体中。比如，在所有曾经存在过的人之外，存在着一个原型的人类，我们之所以是人，是因为我们与那个原型相似；在世界上所有强壮的事物之外，存在着最终的力量概念，所有强壮的事物都从中借取了强壮的属性等等。虽然作为凡人，我们生活在表象的世界中，无法感知形式的世界，但通过哲学，我们可以“回忆”这个世界，并认识到其中的一些特征。

以上是希腊哲学家柏拉图提出的世界观的总结，有时被称为柏拉图的形式理论 (theory of forms)。最初，逻辑学代表了通过思考和结构化思想，以某种形式化方式应用于这个形式世界的努力。今天，这种最初的逻辑范式被称为“经典逻辑” (classical logic)。虽然这一切都始于柏拉图，但大部分功劳要归于20世纪的数学家大卫·希尔伯特 (David Hilbert)。

形式世界的存在意味着，即使我们人类有很多不知道的事情，甚至永远不会知道，至少在某个地方，每一个问题都有一个答案。在逻辑学中，这可以被解释为二值原则 (the principle of bivalence)，即每个命题要么为真，要么为假。由于这一原则，经典逻辑中的命题可以通过集合论中的布尔集来表示，该布尔集包含这两个值。



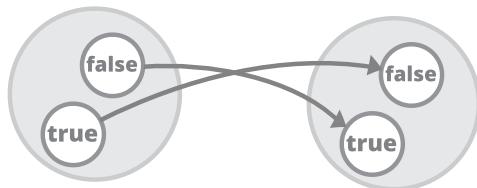
The set of boolean values

根据经典解释, 你可以把基本命题 (primary propositions) 视为一堆布尔值。逻辑运算 (logical operators) 是将一个或多个布尔值作为输入并返回另一个布尔值的函数(而复合命题就是这些函数应用的结果)。

让我们在这个语义背景下回顾所有的逻辑运算。

否定运算 (The *negation* operation)

首先来看否定 (negation) 运算。否定是一个一元运算, 这意味着它是一个只接受一个参数的函数, 并且(像所有其他逻辑运算一样)返回一个布尔值, 该参数和返回值都为布尔值。



negation

同样的函数也可以用这个稍微不那么复杂的表格来表示:

p	$\neg p$
True	False

p	$\neg p$
False	True

类似这样的表格被称为真值表 (truth tables), 它们在经典逻辑中无处不在。它们不仅可以用于定义运算符, 还可以用于证明结果。

插曲: 通过真值表证明结果 (Interlude: Proving results by truth tables)

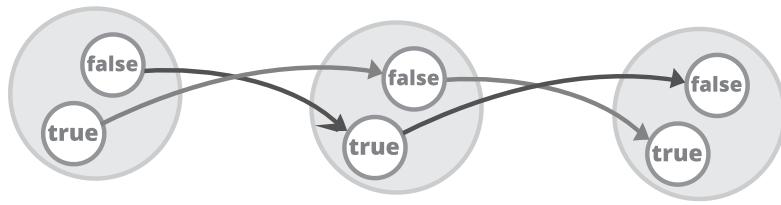
定义了否定运算符之后, 我们现在可以证明逻辑系统中的第一个公理, 即双重否定消去 (double negation elimination)。在自然语言中, 这个公理等价于这样的观察: 说“我不不能做X”与说“我能做X”是一样的。



Double negation elimination formula

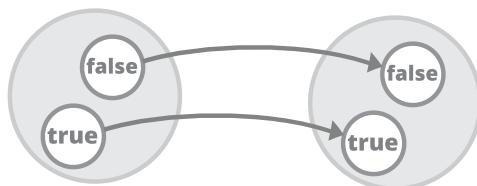
(尽管双重否定公理看起来很简单, 但它可能是逻辑学中最具争议的结果, 原因我们稍后会看到。)

如果我们将逻辑运算符视为从布尔值集到布尔值集的函数, 那么证明公理的过程就是将几个这样的函数组合成一个, 并观察其输出。具体来说, 上述公式的证明只涉及将否定函数与自身组合, 并验证它是否将我们带回原点。



Double negation elimination

如果我们想要正式一些，可以说应用两次否定运算等同于应用恒等(identity)函数。



The identity function for boolean values

如果我们对图表感到厌倦了，我们还可以将上面的组合图表表示为一个表格：

p	$\neg p$	$\neg\neg p$
True	False	True
False	True	False

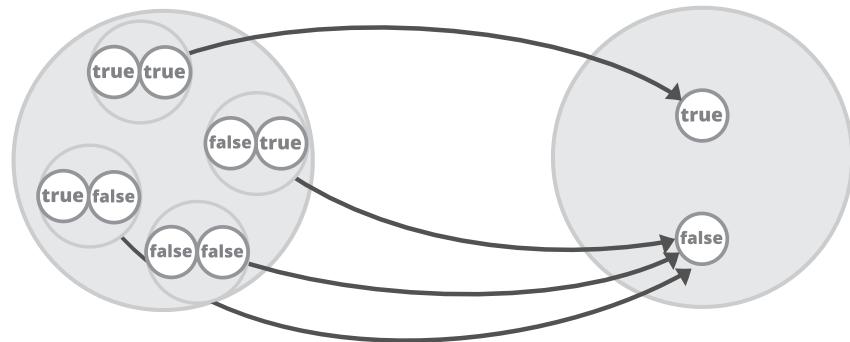
在经典逻辑中，每个命题都可以通过这样的图表或表格来证明。

与和或运算 (The *and* and *or* operations)

好吧，你知道与(*and*)运算是什么意思，我也知道它的意思，但那些想要一切都正式指定的人呢？(提醒一下！) 嗯，我们已经知道如何让

他们满意——我们只需构造表示“与”的布尔函数。

因为与是一个二元运算符，所以该函数接受一对布尔值作为输入。



And

以下是真值表(\wedge 是表示与的符号)：

p	q	$p \wedge q$
True	True	True
True	False	False
False	True	False
False	False	False

我们可以对或(or)进行类似操作，下面是对应的真值表：

p	q	$p \vee q$
True	True	True
True	False	True
False	True	True
False	False	False

任务：绘制或(or)的图表。

利用这些表格，我们还可以证明一些可以稍后使用的公理模式：

- 对于与, $p \wedge q \rightarrow p$ 和 $p \wedge q \rightarrow q$ (“如果我累了又饿，这意味着我饿了”）。

- 对于或, $p \rightarrow p \vee q$ 和 $q \rightarrow p \vee q$ (“如果我有一支笔, 这意味着我有一支笔或尺子”）。

蕴含运算 (The *implies* operation)

现在让我们看一些不那么简单的东西：蕴含运算 (implies operation), 也称为材料条件 (material condition)。这个运算将两个命题联系起来，使得第一个命题的真实性意味着第二个命题的真实性（或者说，第一个命题是第二个命题的必要条件）。你可以将 $p \rightarrow q$ 理解为“如果 p 为真，那么 q 必须也为真”。

蕴含运算也是一个二元函数——它表示从一个有序布尔值对到布尔值的函数。

p	q	$p \rightarrow q$
True	True	True
True	False	False
False	True	True
False	False	True

现在有一些不太显而易见的方面，让我们逐一说明：

1. 如果 p 为真且 q 也为真，则 p 确实蕴含 q ——显然如此。
2. 如果 p 为真但 q 为假，那么 q 并不从 p 得出——因为如果 q 从 p 得出， q 就应该为真。
3. 如果 p 为假而 q 为真， p 仍然蕴含 q 。这是什么意思呢？请考虑，声明 p 蕴含 q 并不意味着两者之间是完全依赖的，例如“喝酒会导致头痛”的说法并不意味着头痛的唯一来源是喝酒。
4. 最后，如果 p 为假且 q 也为假， p 仍然蕴含 q （只是换了一个场景）。

你可以记住，在经典逻辑中， $p \rightarrow q$ (p 蕴含 q) 为真当且仅当 $\neg p \vee q$ (要么 p 为假，要么 q 为真)。

当且仅当运算 (The *if and only if* operation)

现在，让我们回顾一下指示两个命题等价的运算（即当一个命题是另一个命题的必要且充分条件时——这意味着逆命题也为真）。当两个命题具有相同的真假值时，此运算返回真值。

$p \quad q \quad p \leftrightarrow q$

True True True

True False False

False True False

False False True

但更有趣的是，这个运算可以通过蕴含运算构造出来——它等价于每个命题蕴含另一个命题（因此 $p \leftrightarrow q$ 相当于 $p \rightarrow q \wedge q \rightarrow p$ ），这可以通过比较真值表轻松证明。

$p \quad q \quad p \rightarrow q \quad q \rightarrow p \quad p \rightarrow q \wedge q \rightarrow p$

True True True True True

True False False True False

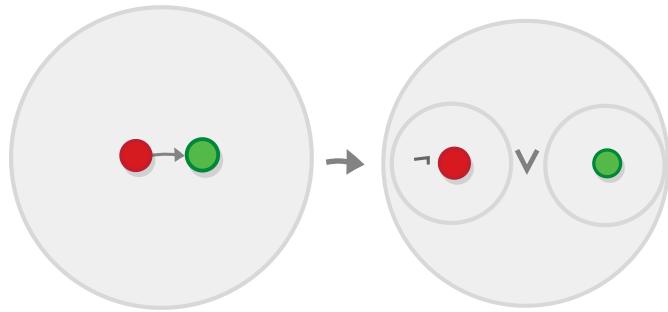
False True True False False

False False True True True

正因为如此，该等价运算被称为“当且仅当”(if and only if)，缩写为“iff”。

通过公理/推理规则证明结果 (Proving results by axioms/rules of inference)

让我们看看上述公式，指出 $p \rightarrow q$ 与 $\neg p \vee q$ 是等价的。



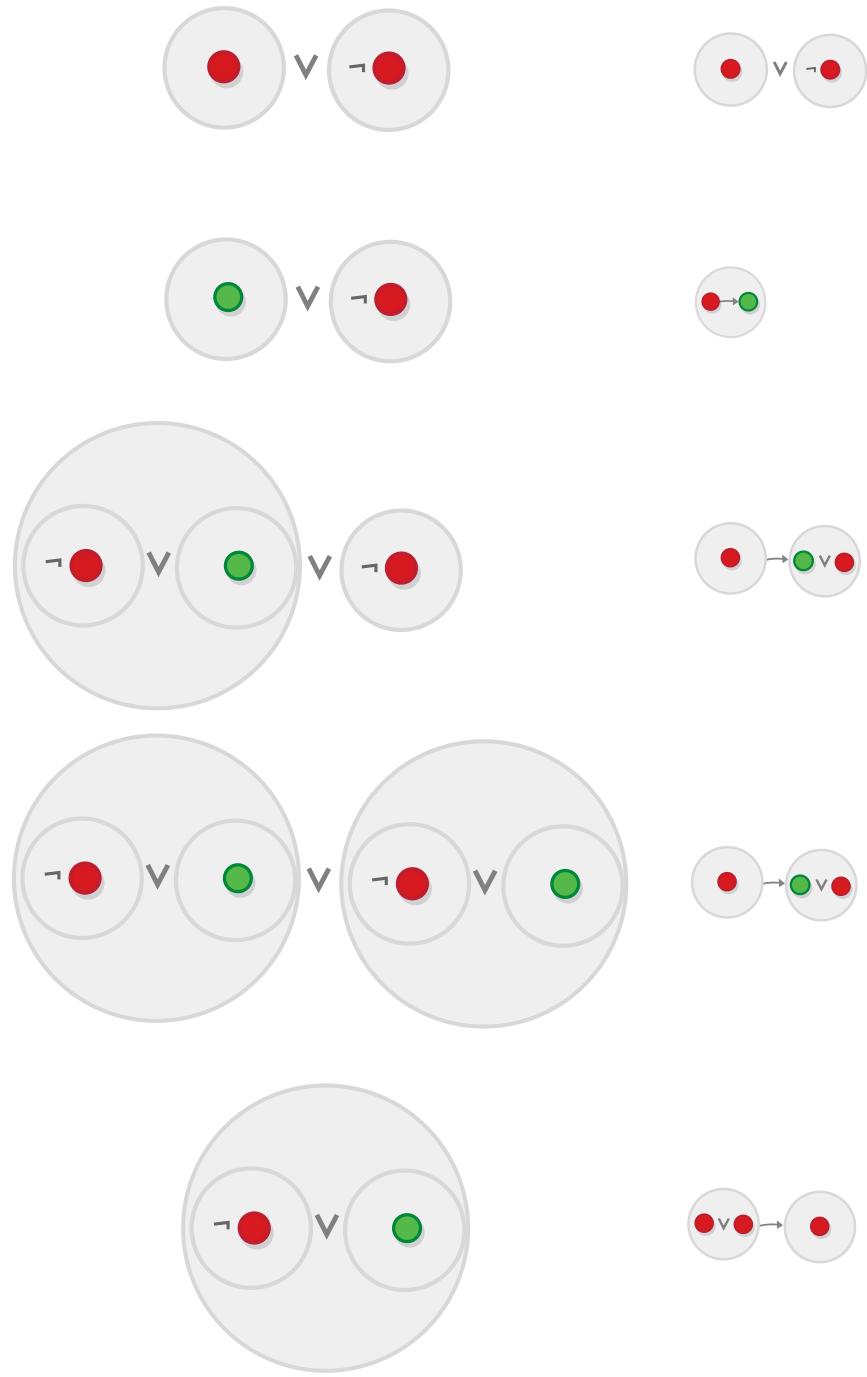
Hilbert formula

我们可以轻松地使用真值表来证明这一点。

p	q	$p \rightarrow q$	$\neg p$	q	$\neg p \vee q$
True	True	True	False	True	True
True	False	False	False	False	False
False	True	True	True	True	True
False	False	True	True	False	True

但如果我们使用公理和推理规则来证明这一点，会更直观。为此，我们从已知公式 $p \rightarrow q$ 开始，加上公理模式，最后得出我们想要证明的公式 $\neg p \vee q$ 。

以下是一种可能的推理过程。每一步使用的公式在右侧标注，推理规则为推理规则。



Hilbert proof

需要注意的是,要真正证明这两个公式是等价的,我们还需要反向证明(即从 $\neg p \vee q$ 开始推导 $p \rightarrow q$)。

直觉主义逻辑 (Intuitionistic logic). BHK 解释 (The BHK interpretation)

[...] 逻辑是人脑中的生活；它可能伴随脑外的生活，但永远不能凭借自身的力量指导它。—— L.E.J. 布劳威尔

我不知道你是怎么想的，但我觉得经典的真值函数解释（虽然它在自身的领域内有效并且是正确的）并不太适合我们在这里使用的范畴框架：它太“低级”了，它依赖于对命题的值进行操作。根据这种解释，和 (and) 和 或 (or) 操作只是16种可能的二元逻辑操作中的两种，它们之间没有真正的联系（但我们知道它们实际上是有相互关联的）。

基于这些以及其他原因，20世纪诞生了一整个新的逻辑学派，称为直觉主义逻辑 (intuitionistic logic)。如果我们将经典逻辑视为基于集合论 (set theory)，那么直觉主义逻辑则基于范畴论 (category theory) 及其相关理论。如果说经典逻辑是基于柏拉图的理念论 (Plato's theory of forms)，那么直觉主义起源于康德 (Kant) 和叔本华 (Schopenhauer) 的哲学思想：即我们所体验的世界在很大程度上由我们的感知预先决定。因此，在没有绝对真理标准的情况下，命题的证明成为你构造出来的东西，而不是你发现的东西。

经典逻辑和直觉主义逻辑从一开始就分道扬镳：因为根据直觉主义逻辑，我们是构造证明，而不是像某种普遍真理那样发现证明，我们就偏离了二值性原则 (the principle of bivalence)。也就是说，在直觉主义逻辑中，我们没有依据宣称每个命题必定为真或为假。例如，某些命题可能无法被证明，这并不是因为它们是假的，而是因为它们超出了给定逻辑系统的范围（孪生素数猜想经常被用作例子）。

总之，直觉主义逻辑不是二值的，即我们不能将所有命题简化为真和假。



真/假二分法

有一件事我们依然有，那就是某些命题是“真的”，意味着有一个证明可以为它们提供支持——这是基本命题。所以，在某些限制条件下（稍后我们会看到），可以将真命题与假命题之间的二值性视为命题的证明存在或不存在之间的二值性——要么有证明，要么没有。



已证明/未证明二分法

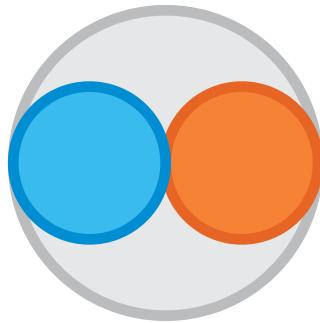
这种二值性是所谓的布劳威尔–海廷–柯尔莫戈罗夫 (Brouwer–Heyting–Kolmogorov, BHK) 逻辑解释的核心，我们接下来会深入探讨。

BHK 解释的原始形式并不基于任何特定的数学理论。这里，我们将首先用集合论的语言来说明它（只是为了稍后将其放弃）。

和和 或 操作

由于命题的证明存在意味着命题为真，因此 和 (and) 的定义非常简单—— $A \wedge B$ 的证明就是一个包含 A 的证明和 B 的证明的对，即它们的集合论乘积（参见第2章）。决定命题真假的原则与基本命题相似——如果

A 和 B 的证明对存在(即, 如果两个证明都存在), 那么 $A \wedge B$ 的证明可以被构造(因此 $A \wedge B$ 是“真”的)。

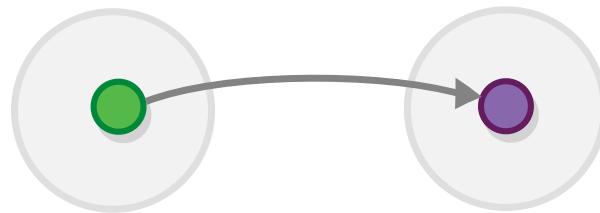


BHK 解释中的和操作

问题：在这种情况下, 或 操作会是什么?

蕴含操作

现在到了关键点: 在 BHK 解释中, 蕴含操作只是证明之间的函数。说 A 蕴含 B ($A \rightarrow B$) 只意味着存在一个函数可以将 A 的证明转换为 B 的证明。



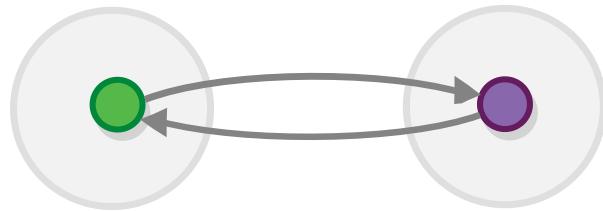
BHK 解释中的蕴含操作

更有趣的是, 蕴含规则(modus ponens) 只不过是函数应用的过程。即, 如果我们有 A 的证明和一个 $A \rightarrow B$ 的函数, 我们可以调用该函数来得到 B 的证明。

(为了正式定义这一点, 我们还需要以集合的形式定义函数, 即我们需要为每个 A 和 B 具有一个表示 $A \rightarrow B$ 的集合。稍后我们会回到这个话题。)

当且仅当操作

在经典逻辑部分, 我们证明了当 A 蕴含 B 且 B 蕴含 A 时, 两个命题 A 和 B 是等价的。但如果蕴含操作只是一个函数, 那么命题等价的条件恰好是存在两个函数将它们相互转换, 即命题的集合是同构的。



BHK 解释中的当且仅当操作

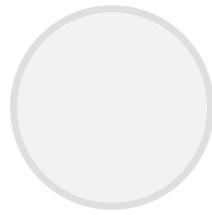
(或许我们应该注意到, 并不是所有的集合论函数都是证明, 只有一个指定的函数集合(我们称之为规范函数), 即在集合论中你可以在任意两个单元素集合之间构造函数和同构, 但这并不意味着所有证明都是等价的。)

否定操作

因此, 根据 BHK 解释, 说 A 为真意味着我们拥有 A 的证明——这很简单。但要表达 A 为假就有点困难了: 仅仅说我们没有证明 A 是不够的(没有证明并不意味着它不存在)。相反, 我们必须表明, 声称 A 为真会导致矛盾。

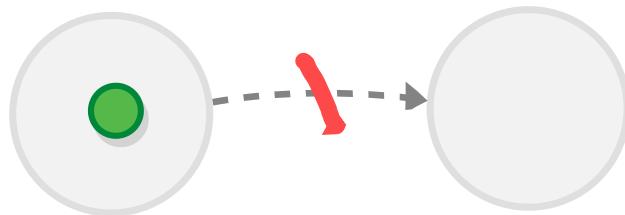
为了表达这一点, 直觉主义逻辑定义了常量 \perp , 它起到假的作用(也称为“底部值”)。 \perp 被定义为一个没有任何证明的公式的证明。而假命题则是那些蕴含底值可证明的命题(这是一种矛盾)。因此, $\neg A$ 就是 $A \rightarrow \perp$ 。

在集合论中, \perp 常量通过空集来表示。



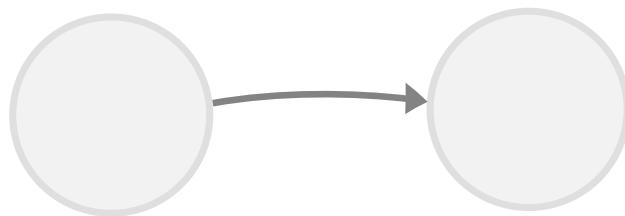
BHK 解释中的假操作

命题与底值相连的观察则通过这样表达:如果一个命题为真,即存在它的证明,那么不可能存在从该命题到空集的函数。



BHK 解释中的假函数

唯一能够存在此类函数的情况是该命题的证明集合也是空的。



BHK 解释中的假函数2

任务: 查找函数的定义,并验证不能存在从任何集合到空集的函数。

任务: 查找函数的定义,并验证确实存在从空集到自身的函数(实际上存在从空集到任何其他集合的函数)。

排中律

虽然直觉主义逻辑与经典逻辑在语义上(即系统的构建方式)有很大不同,但在语法上它们并没有太多差异。也就是说,如果我们尝试推导出与上述结构定义相对应的公理模式/推理规则,会发现它们与定义经典逻辑的规则几乎相同。然而,存在一个例外,就是我们之前看到的双重否定消除公理(double negation elimination axiom),其中一个版本被称为排中律(the law of excluded middle)。



排中律公式

该定律在经典逻辑中是有效的,并且在真值表中为真,但在BHK解释中没有依据。为什么?在直觉主义逻辑中,说某事为假等于构造一个证明来表明它为假(即它蕴含底值),而不存在任何可以证明给定命题为真或假的方法/函数/算法。

关于能否使用排中律的问题引发了一场激烈的辩论,经典逻辑的支持者大卫·希尔伯特(David Hilbert)与直觉主义逻辑的支持者L.E.J.布劳威尔(L.E.J. Brouwer)展开了争论,这场争论被称为布劳威尔-希尔伯特争论(the Brouwer–Hilbert controversy)。

逻辑作为范畴 (Logics as categories)

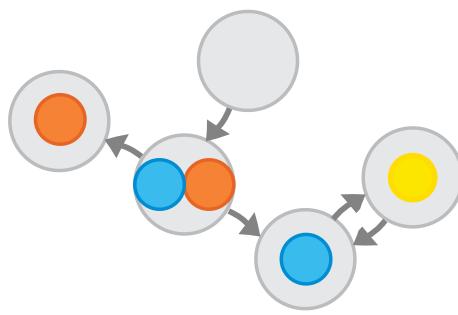
撇开直觉主义逻辑和经典逻辑之间的差异不谈

, BHK 解释的有趣之处在于它提供了一种更高层次的逻辑视角, 这是我们基于范畴论构建逻辑解释所需要的。

这种更高层次的逻辑解释有时被称为代数解释, 代数是一个广义术语, 描述了所有可以用范畴论表示的结构, 比如群和序。

柯里-霍华德同构 (The Curry-Howard isomorphism)

程序员可能会觉得 BHK 解释的定义很有趣, 因为它与编程语言的定义非常相似: 命题是类型, 蕴含操作是函数, 和操作是复合类型(对象), 而或操作是和类型(目前大多数编程语言还不支持这种类型, 但这是另一个话题)。最终, 命题的证明通过对应类型的值来表示。



这种相似性被称为柯里-霍华德同构 (the Curry-Howard isomorphism)。

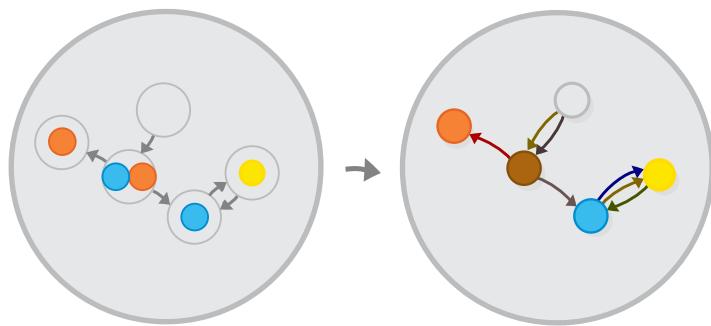
任务: 柯里-霍华德同构也是一种特殊类型编程语言(称为“证明助手”)的基础, 它可以帮助你验证逻辑证明。安装一个证明助手并尝试使用它

(我推荐 Mike Nahas 的 Coq 教程)。

笛卡尔闭范畴 (Cartesian closed categories)

知道了柯里–霍华德同构，并且也知道编程语言可以用范畴论来描述，可能会让我们认为范畴论也是该同构的一部分。我们的想法是正确的——这就是为什么它有时被称为柯里–霍华德–兰贝克 (Curry-Howard-Lambek) 同构(约阿希姆·兰贝克 (Joachim Lambek) 是提出范畴方面同构的人)。让我们来研究一下这种同构。像其他同构一样，它由两部分组成：

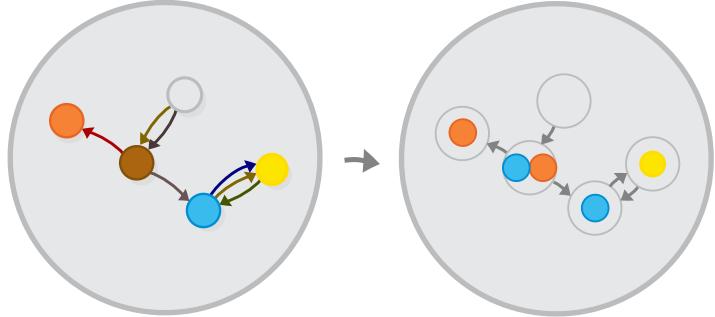
第一部分是将逻辑系统转换为范畴——这对我们来说并不难，因为集合构成了一个范畴，而我们看到的 BHK 解释的形式是基于集合的。



逻辑作为范畴

任务：看看你能否证明逻辑命题和“蕴含”关系构成一个范畴。还缺少什么？

第二部分涉及将范畴转换为逻辑系统——这要困难得多。为了做到这一点，我们必须列举一个范畴必须遵守的标准，以使其成为“逻辑的”。这些标准必须保证范畴中有对应于所有有效逻辑命题的对象，而没有对应于无效命题的对象。

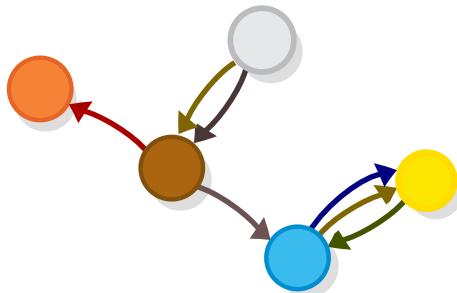


逻辑作为范畴

符合这些标准的范畴被称为笛卡尔闭范畴 (cartesian closed categories)。我们将不会直接描述它们，而是从我们已经研究过的类似但更简单的结构——序开始。

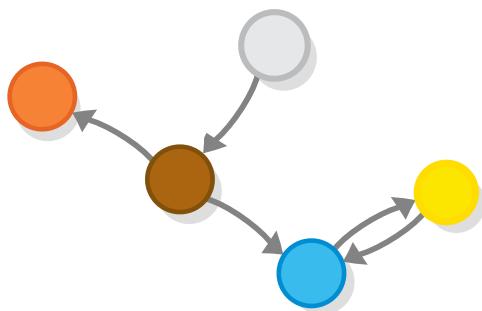
逻辑作为序 (Logics as orders)

所以，我们已经看到一个逻辑系统和一组基本命题形成了一个范畴。



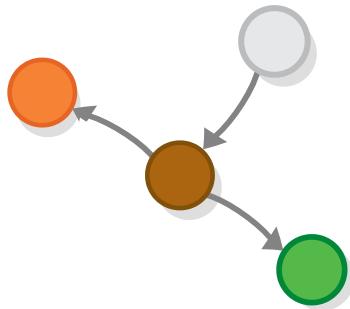
逻辑作为预序

如果我们假设从命题 A 到命题 B 只有一种方式（或者有很多方式，但我们对它们之间的差异不感兴趣），那么逻辑不仅是一个范畴，而且是一个预序，其中“比……大”的关系意味着“蕴含”，因此 $(A \rightarrow B \text{ 是 } A > B)$ 。



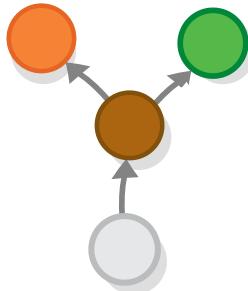
逻辑作为预序

此外,如果我们将彼此相互推导的命题(或由同一证明证明的一组命题)视为等价,那么逻辑就是一个真正的偏序。



逻辑作为序

因此它可以用一个哈斯图 (Hasse diagram) 来表示,其中 $A \rightarrow B$ 仅当 A 位于图中的 B 之下。



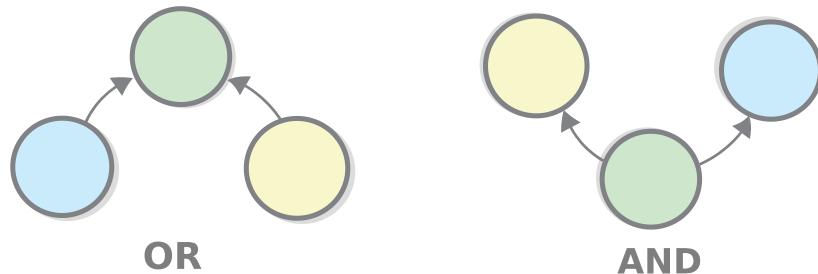
逻辑作为序

这是范畴论的一个非常典型的特征——在一个更有限的范畴版本中检查一个概念(在本例中是序),以简化我们的分析。

现在让我们重新审视之前提出的问题——究竟哪些范畴序表示逻辑，序必须遵守哪些定律才能与逻辑系统同构？我们将在重新讨论逻辑元素时，在序的背景下尝试回答这个问题。

和和或操作

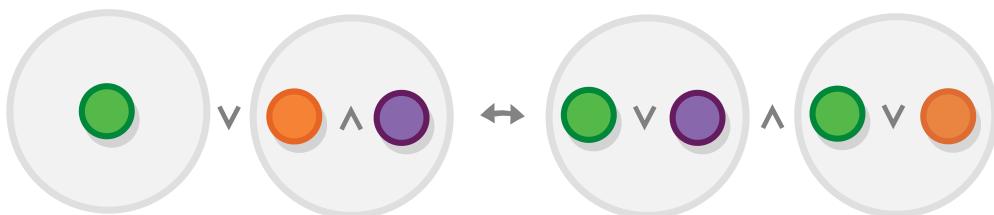
到现在为止，你可能已经意识到和 (and) 和 或 (or) 操作是逻辑的基本组成部分（尽管还不清楚哪一个是“和”，哪一个是“或”）。正如我们所见，在 BHK 解释中，它们由集合的积 (products) 和 和 (sums) 表示。在序理论中，对应的结构是交 (meets) 和并 (joins)（在范畴论术语中，分别是积和余积）。



序的交和并

逻辑允许你将任意两个命题结合在和或或关系中，因此，为了使序“逻辑化”（即成为逻辑系统的正确表示），它必须对所有元素具有 *meet* 和 *join* 操作。顺便说一句，我们已经知道这种序叫什么——它们叫做格 (lattices)。

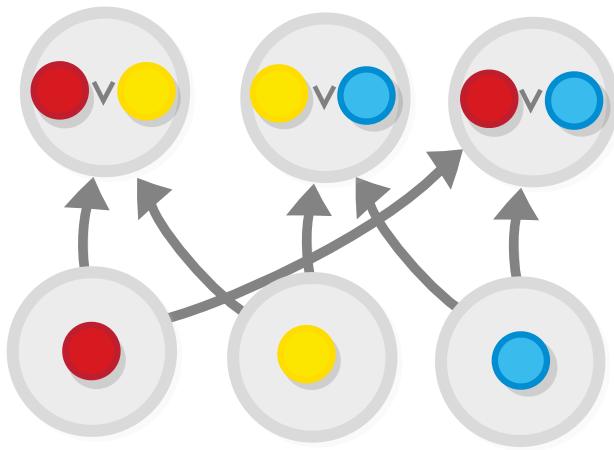
和和或操作有一条重要定律，它并不总是出现在所有格中。它涉及二者之间的连接方式，即它们在彼此上的分布方式。



“和”和“或”的分配律

遵守该定律的格被称为分配格 (distributive lattices)。

等等, 我们之前在哪里听过分配格? 在前一章中, 我们说它们与包含序 (inclusion orders) 同构, 即包含给定元素集的序, 并且包含这些元素集的所有组合。它们再次出现并非偶然——“逻辑的”序与包含序同构。要理解原因, 你只需要考虑 BHK 解释——参与包含的元素就是我们的基本命题。包含就是这些元素的所有组合, 以或关系的形式(为了简化起见, 我们忽略了和操作)。



颜色混合偏序, 按包含排序

或 和 和 操作(或更一般地, 余积 和 积)在范畴上是对偶的, 这也可以解释为什么表示它们的符号 \vee 和 \wedge 是同一个符号, 只不过垂直翻转。

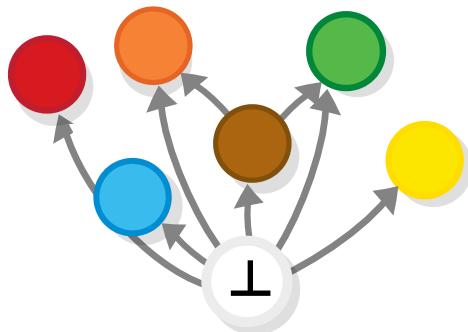
甚至连符号本身看起来也像箭头汇聚的表示。这可能并非巧合, 因为这种符号的使用远早于哈斯图的出现——据我们所知, \vee 符号可能象征着拉丁词“uel”(“或”)中的“u”, 而和符号则只是一个倒转的“u”——但我仍然觉得这种相似性很有趣。

否定操作

为了使分配格表示一个逻辑系统, 它还必须有对应真 (True) 和 假 (False) 值的对象 (分别写作 \top 和 \perp)。但是, 要规定这些对象的存

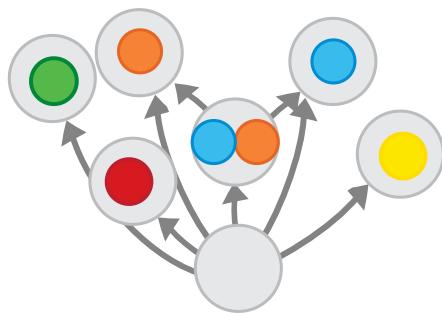
在，我们必须首先找到一种方式来指定它们在序/范畴论术语中的定义。

逻辑中一个著名的结果，称为爆炸原理 (the principle of explosion)，指出如果我们有一个假的证明 (我们写作 \perp)，即如果我们有一个“假为真”的陈述 (使用经典逻辑的术语)，那么任何其他陈述都可以被证明。而且我们也知道没有一个真实陈述能蕴含假 (事实上，在直觉主义逻辑中，这是定义一个真陈述的方式)。基于这些标准，我们知道假对象在与其他对象的比较中如下所示：



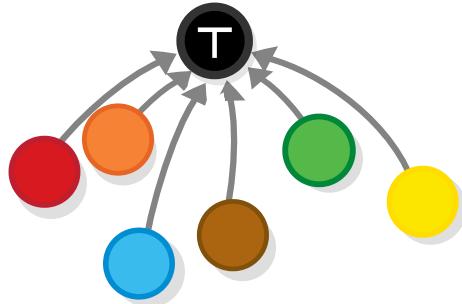
哈斯图中的假

回到 BHK 解释，我们看到空集符合这两个条件。



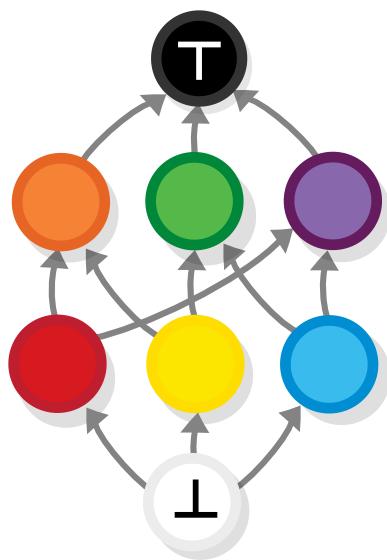
哈斯图中的假

相反，表示真 (True) 的证明，我们写作 \top ，表达的是“真为真”这一陈述，这是微不足道的，并没有任何信息，因此没有任何东西能从它推导出来，但同时它又从每个其他陈述中推导出来。



哈斯图中的真

因此真和假只是我们序的最大和最小对象（在范畴论术语中，分别是终端 (terminal) 和初始 (initial) 对象）。



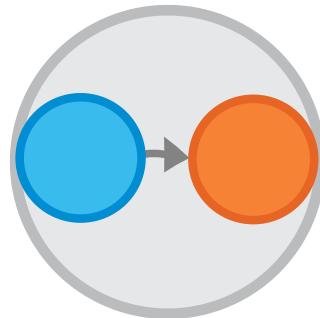
整个逻辑系统的哈斯图表示

这是范畴论中对偶性的另一个例子—— \top 和 \perp 是相互对偶的，这很有意义，如果你仔细思考的话，也有助于我们记住它们的符号（尽管如果你像我一样，每次看到它们时总是要花一年的时间来搞清楚哪个是哪个）。

因此，为了表示逻辑，我们的分配格还必须是有界的 (bounded)，即它必须具有最大和最小元素（分别扮演真和假的角色）。

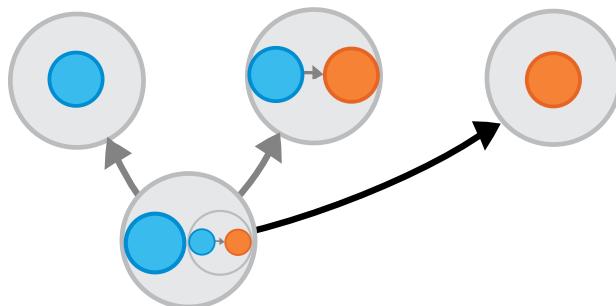
蕴含操作

正如我们所说,每个格都有表示命题相互蕴含的表示(即它具有箭头),但要真正表示一个逻辑系统,它还必须有函数对象(function objects),即必须有一个规则为每对对象 A 和 B 识别一个唯一的对象 $A \rightarrow B$,以便遵循所有逻辑公理。



蕴含操作

我们将像定义其他操作那样定义该对象——通过定义一个由对象和箭头组成的结构,其中 $A \rightarrow B$ 扮演重要角色。该结构实际上是我们最喜欢的推理规则蕴含规则(modus ponens)的范畴重现。

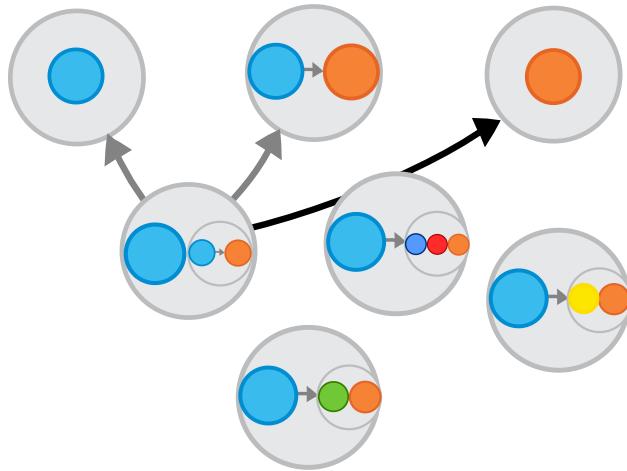


蕴含操作

蕴含规则是蕴含操作的本质,而且因为我们已经知道它所包含的操作(和和蕴含)如何在我们的格中表示,所以我们可以直接将其用作定义,声明对象 $A \rightarrow B$ 是满足蕴含规则的对象。

函数对象 $A \rightarrow B$ 是一个与对象 A 和 B 相关的对象,使得 $A \wedge (A \rightarrow B) \rightarrow B$ 。

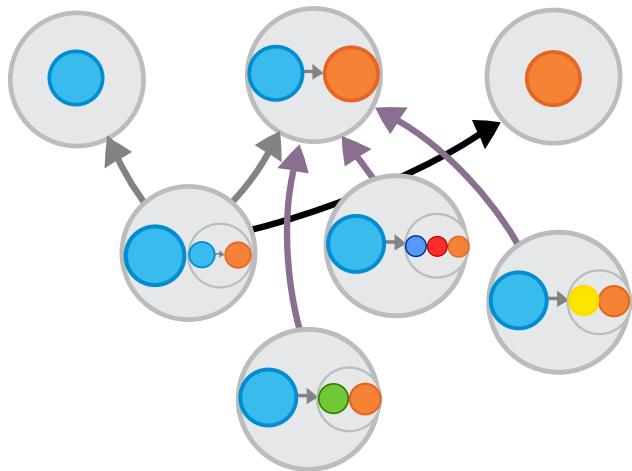
然而,这个定义是不完整的,因为(如同往常一样) $A \rightarrow B$ 并不是唯一满足这个公式的对象。例如,集合 $A \rightarrow B \wedge C$ 也是这样的一个对象, $A \rightarrow B \wedge C \wedge D$ 也是(我不会在这里画出所有的箭头,因为它会变得过于(真的很过于)复杂)。



具有普遍性质的蕴含操作

那么我们如何区分真正的公式和所有这些“冒牌”公式呢?如果你记得范畴积(categorical product)的定义(或它在序中的等价物,即交操作),你已经知道我们将要去的地方:我们认识到 $A \rightarrow B$ 是 $A \rightarrow B \wedge C$ 和 $A \rightarrow B \wedge C \wedge D$ 以及所有其他冒牌公式的上限(upper limit)。这种关系可以通过多种方式描述:

- 我们可以说 $A \rightarrow B$ 是使公式 $A \wedge X \rightarrow B$ 成立的最平凡的结果,所有其他结果都更强。
- 我们可以说所有其他结果都蕴含 $A \rightarrow B$,但反之不然。
- 我们可以说所有其他公式都在哈斯图中低于 $A \rightarrow B$ 。



具有普遍性质的蕴含操作

所以, 在选择表达关系的最佳方式之后(它们都是等价的), 我们准备给出我们的最终定义:

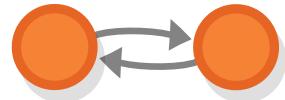
函数对象 $A \rightarrow B$ 是满足公式 $A \wedge (A \rightarrow B) \rightarrow B$ 的最高对象。

该函数对象 (在范畴论术语中称为 **指数对象** (exponential object)) 的存在是序/格成为逻辑表示的最后一个条件。

顺便提一下, 这种函数对象的定义仅对直觉主义逻辑有效。对于经典逻辑, 定义更简单——在经典逻辑中, $A \rightarrow B$ 只是 $\neg A \vee B$, 因为排中律的存在。

当且仅当 操作

当我们检查当且仅当 (if and only if) 操作时, 我们看到它可以用蕴含来定义, 即 $A \leftrightarrow B$ 等价于 $A \rightarrow B \wedge B \rightarrow A$ 。



蕴含等价

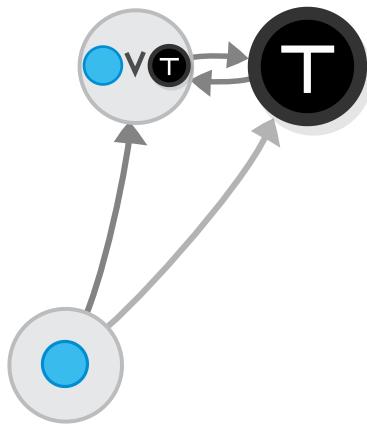
在范畴逻辑中，我们也有类似的东西——我们说当两个命题相互关联时，特别是在我们谈论序时，它们是同构的。

范畴逻辑的初体验 (A taste of categorical logic)

在上一节中, 我们看到了许多定义, 在这里我们将通过范畴逻辑证明一些结果, 以验证这些定义是否确实正确地捕捉了逻辑的概念。

真和假

上限 (join) (或最小上界 (least upper bound)) 是对象 \top (起到真的作用) 和任何你能想到的其他对象的结合结果是…… \top 本身 (或与它同构的东西, 如我们所说, 这就是同一件事)。这一点可以从以下事实中推导出来: 两个对象的上限必须大于或等于这两个对象, 并且没有其他对象比 \top 更大。因此唯一可能的上限就是 \top 本身。这是因为 \top (与任何其他对象一样) 等于它自身, 并且根据定义, 没有其他对象比它更大。



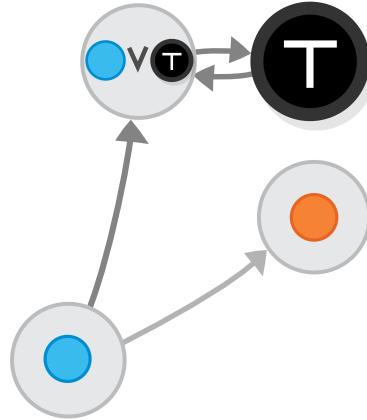
蕴含等价

这对应于逻辑陈述 $A \vee \top$ 等于 \top , 即为真。因此, 上述观察是该陈述的证明 (作为真值表的替代方案)。

任务: 想想与假相对的情况, 它在逻辑上意味着什么?

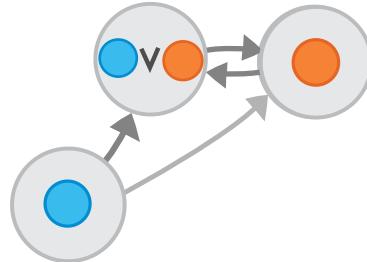
和和或

上面我们看到，任何随机对象与最大对象的上限是最大对象本身。但这种情况是否只会发生在第二个对象是 \top 时？如果用比第一个对象大的任何其他对象代替 \top ，难道不会发生同样的情况吗？



蕴含等价

答案是“是的”：当我们寻找两个对象的上限时，我们寻找的是最小上界，即位于它们两者之上的最低对象。因此，任何时候我们有两个对象且一个对象比另一个高时，它们的上限将（与）更高的对象（同构）。

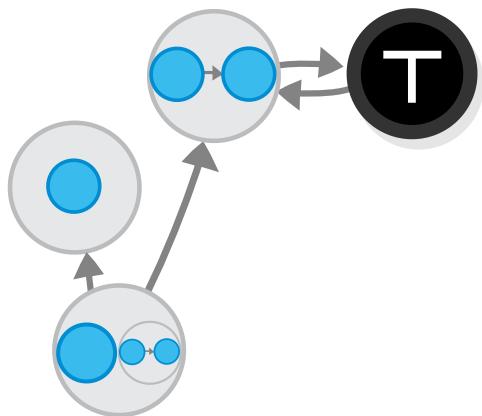


蕴含等价

换句话说，如果 $A \rightarrow B$ ，那么 $A \wedge B \leftrightarrow B^\circ$ 。

蕴含

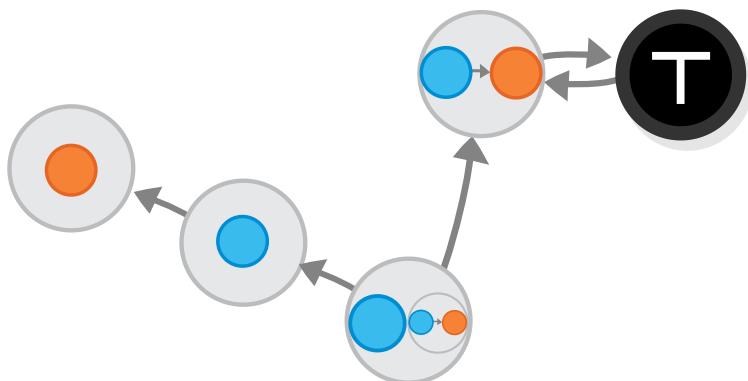
对于我们的第一个蕴含操作示例，让我们以公式 $A \rightarrow B$ 为例，研究 A 和 B 是同一个对象的情况。我们说 $A \rightarrow B$ （在我们的例子中是 $A \rightarrow A$ ）是使公式 $A \wedge X \rightarrow B$ 成立的最高对象 X 。但在这种情况下，该公式对任何 X 都成立（因为它等价于 $A \wedge X \rightarrow A$ ，这是总为真的），因此满足它的最高对象是……最大的对象，即（与之同构的）真。



蕴含等价

这有意义吗？当然有：实际上，我们刚刚证明了逻辑中最著名的定律之一（称为亚里士多德的同一律 (the law of identity)），即 $A \rightarrow A$ 总为真，或者说一切都可以从自身推出。

如果 A 蕴含 B 在任何模型中，即 $A \vDash B$ （语义后果），会发生什么？在这种情况下， A 将位于哈斯图中的 B 之下（例如， A 是蓝色球， B 是橙色球）。然后情况与之前有些类似： $A \wedge X \rightarrow B$ 无论 X 是什么都为真（仅仅因为 A 自己已经蕴含了 B ）。因此 $A \rightarrow B$ 将再次对应于真对象。



当 A 推出 B 时的蕴含

这又是逻辑中的一个著名结果（如果我没记错的话，这将是某种推导定理）：如果 $A \vDash B$, 那么陈述 $(A \rightarrow B)$ 将总为真。

函子 (Functors)

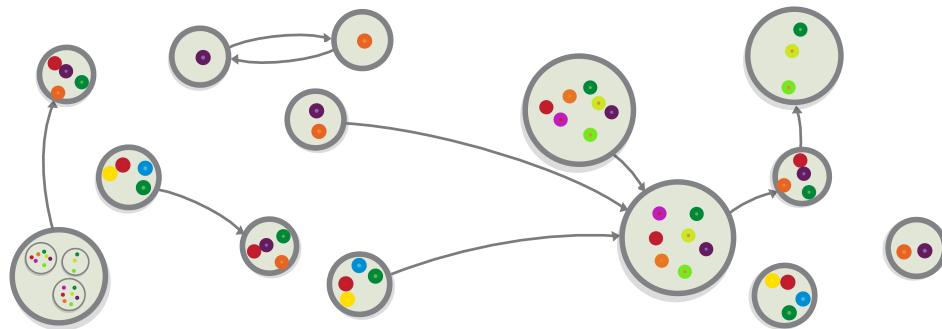
从本章开始, 我们将稍微改变策略(因为我相信你已经厌倦了在不同主题之间跳跃), 我们将全速深入范畴的世界, 使用迄今为止看到的结构作为背景。这将使我们能够推广这些结构中研究过的一些概念, 从而使它们对所有范畴都适用。

我们迄今看到的范畴 (Categories we saw so far)

到目前为止, 我们看到了许多不同的范畴和范畴类型。让我们再回顾一下:

集合范畴 (The category of sets)

我们首先审视了所有范畴之母——集合范畴。

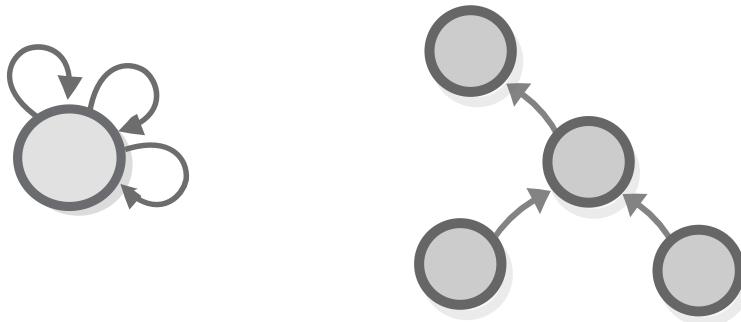


集合范畴

我们还看到了它包含许多其他范畴, 例如编程语言中的类型范畴。

特殊类型的范畴 (Special types of categories)

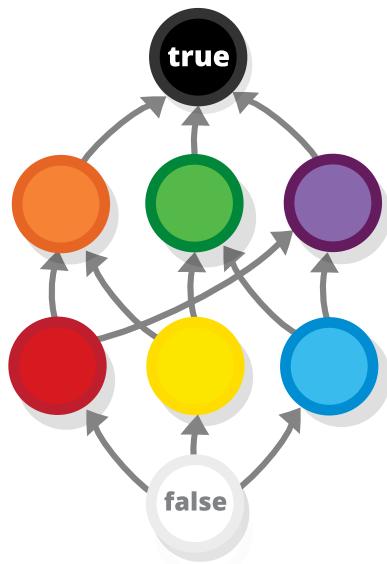
我们还学习了其他代数对象, 它们实际上只是特殊类型的范畴, 例如只包含一个对象的范畴 (如单子、群) 和在任意两个对象之间只有一个态射的范畴 (如预序、偏序)。



范畴类型

其他范畴 (Other categories)

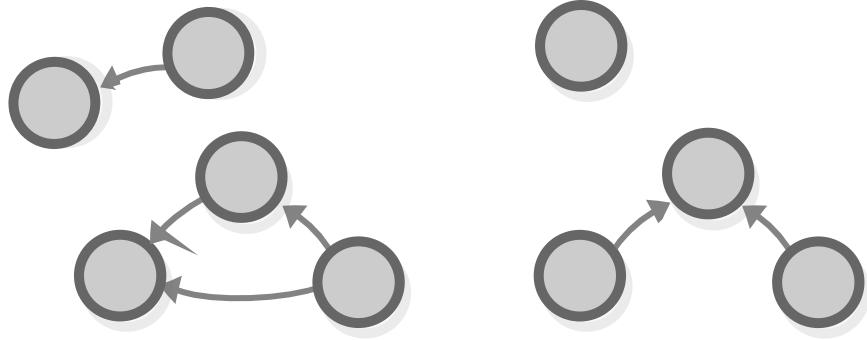
我们还定义了许多基于不同概念的范畴, 例如基于逻辑或编程语言的范畴, 也有一些“没那么严肃的范畴”, 例如颜色混合的偏序范畴。



颜色混合范畴

有限范畴 (Finite categories)

最重要的是, 我们看到了完全是虚构的范畴, 例如我的足球运动员层级。这些正式被称为有限范畴。



有限范畴

尽管它们本身可能没什么用,但背后的想法很重要——我们可以画出任何点和箭头的组合并称其为一个范畴,就像我们可以用任何物体的组合构造一个集合一样。

审视一些有限范畴 (Examining some finite categories)

为将来参考,让我们看看一些重要的有限范畴。

最简单的范畴是 0(享受一下这个简约的图表)。

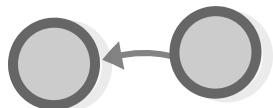
有限范畴 0

接下来是最简单的范畴 1——它由一个对象组成,除了其恒等态射之外没有其他态射(我们不绘制恒等态射,按照惯例)。



有限范畴 1

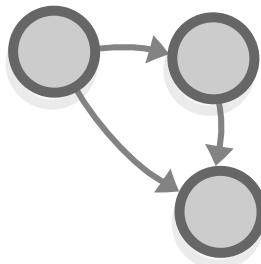
如果我们将对象的数量增加到两个，我们会看到几个更有趣的范畴，例如包含两个对象和一个态射的范畴 2。



有限范畴 2

任务：还有另外两个只包含 2 个对象且两个对象之间至多有一个态射的范畴，画出它们。

最后，范畴 3 包含 3 个对象以及 3 个态射（其中一个是其他两个的复合态射）。



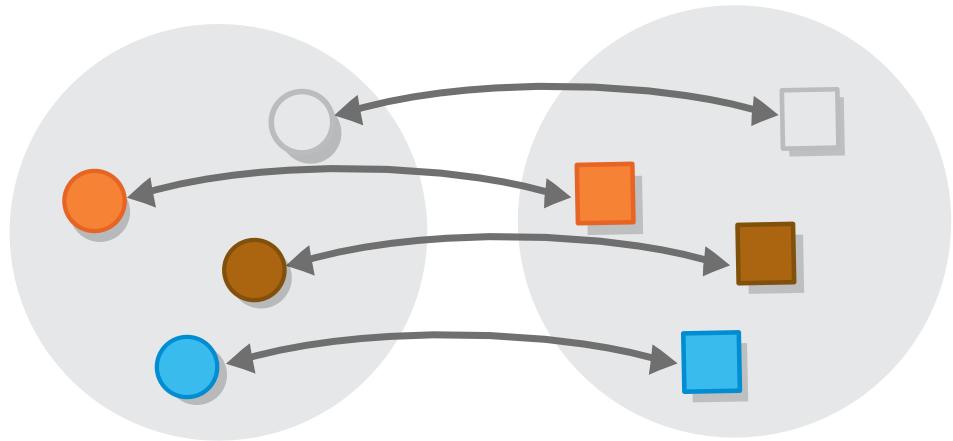
有限范畴 3

范畴同构 (Categorical isomorphisms)

我们所看到的许多范畴彼此非常相似，例如，颜色混合偏序和表示逻辑的范畴都有一个最大和最小对象。为了指出这些相似性并理解它们的意义，能够使用正式的方式将范畴相互连接是很有用的。最简单的一种连接方式就是老朋友同构。

集合同构 (Set isomorphisms)

在第1章中，我们讨论了集合同构，它在两个集合之间建立了等价关系。如果你忘记了，集合同构是集合之间的双向函数。



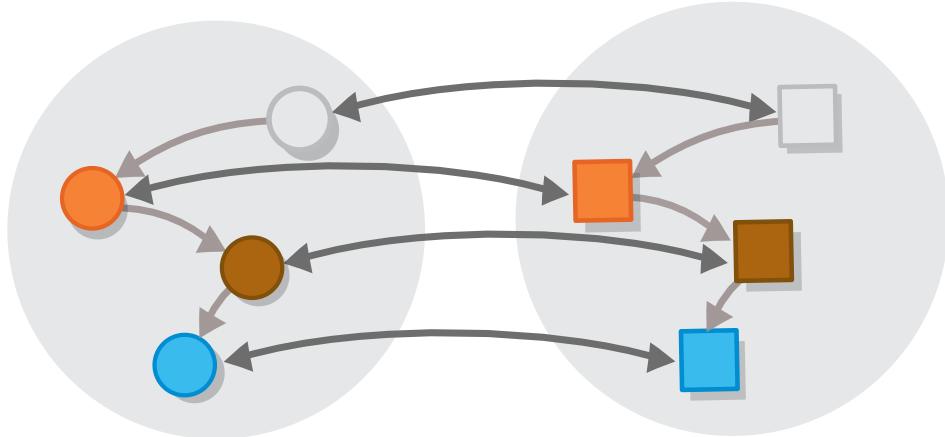
集合同构

它也可以被视为两个“互为逆”的函数，组合后等于恒等函数。

序同构 (Order isomorphisms)

在第4章中，我们遇到了序同构，并且看到它们类似于集合同构，但多了一个条件——除了保持对象之间的映射外，定义同构的函数还必须保持

对象的顺序，例如一个序中的最大对象应该与另一个序中的最大对象连接，最小对象应该与最小对象连接，所有中间的对象也应如此。



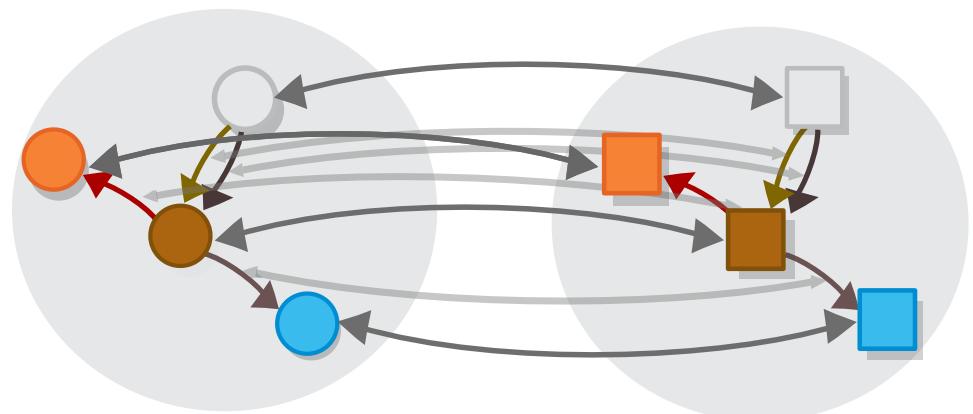
序同构

更正式地说，对于任何 a 和 b ，如果我们有 $a \leq b$ ，那么我们也应该有 $F(a) \leq F(b)$ （反之亦然）。

范畴同构 (Categorical isomorphisms)

现在，我们将推广序同构的定义，使其适用于所有其他范畴（即适用于可能在两个对象之间有多个态射的范畴）：

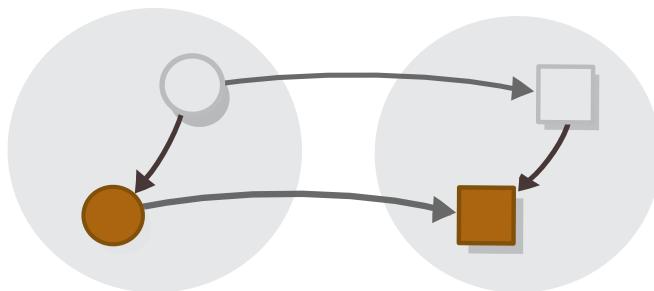
给定两个范畴，它们之间的同构是对象集合之间的可逆映射，以及连接它们的态射之间的可逆映射，并且每个范畴的态射映射到另一个范畴中具有相同签名的态射。



范畴同构

仔细检查这个定义后, 我们会意识到, 尽管它听起来更复杂(看起来更凌乱)一些, 但实际上它与我们对序同构的定义是相同的。

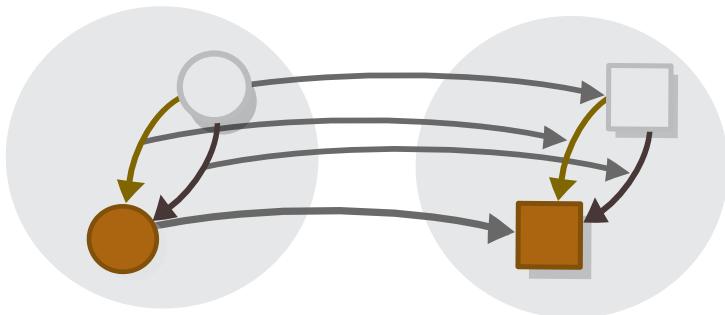
只不过当两个对象之间只有一个态射时, 所谓的“态射映射”是平凡的, 所以我们可以忽略它们。



序同构

问题: 序的态射函数是什么?

然而, 当两个对象之间可以有多个态射时, 我们需要确保源范畴中的每个态射在目标范畴中都有对应的态射。因此, 我们不仅需要对象之间的映射, 还需要它们的态射之间的映射。



范畴同构

顺便说一句, 我们刚刚做的事情(将为更狭窄结构(序)定义的概念重新定义为更广泛的结构(范畴))称为概念推广。

范畴同构的问题 (The problem with categorical isomorphisms)

仔细研究后，我们意识到定义范畴同构并不难。然而，还有另一个问题，即它们并没有捕捉到范畴相等的本质。我想出一个非常好的直观解释，但这段空间太窄，无法容纳这个解释。因此，我们将在下一章中讨论一个更适合定义范畴之间双向连接的方法。

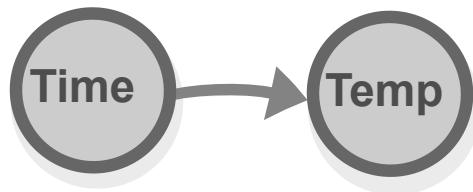
但首先，我们需要研究范畴之间的单向连接，即函子。

PS: 范畴同构在实践中也非常罕见——我能想到的唯一例子是上一章中的柯里-霍华德-兰贝克同构。这是因为如果两个范畴是同构的，那么完全没有理由将它们视为不同的范畴——它们实际上是同一个范畴。

什么是函子 (What are functors)

逻辑学家鲁道夫·卡纳普 (Rudolf Carnap) 首次提出了“函子”这个术语，作为他形式化自然语言(如英语)的项目的一部分，以创建一种精确的方式来讨论科学。最初，函子是指一个单词或短语，它的意义可以通过与数值结合来定制，比如“ x 点的温度”这个短语，其含义根据 x 的值而变化。

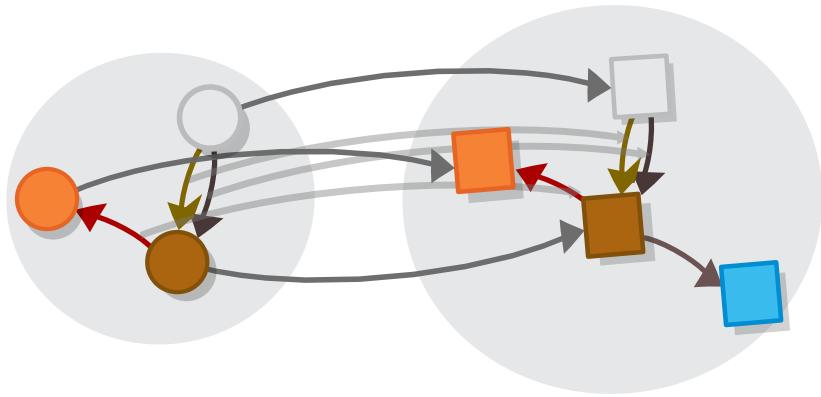
换句话说，函子是一个作为函数的短语，但不是集合之间的函数，而是语言概念之间的函数(例如时间和温度)。



卡纳普设想的函子

后来，范畴论的发明者之一桑德斯·麦克莱恩 (Sanders Mac Lane) 借用了这个词，用来描述在范畴之间充当函数的东西，他将其定义如下：

两个范畴之间的函子 (我们称它们为 A 和 B) 由两个映射组成——一个将 A 中的每个对象映射到 B 中的一个对象，另一个将 A 中任意两个对象之间的每个态射映射到 B 中的态射，这种映射方式保持了范畴的结构。

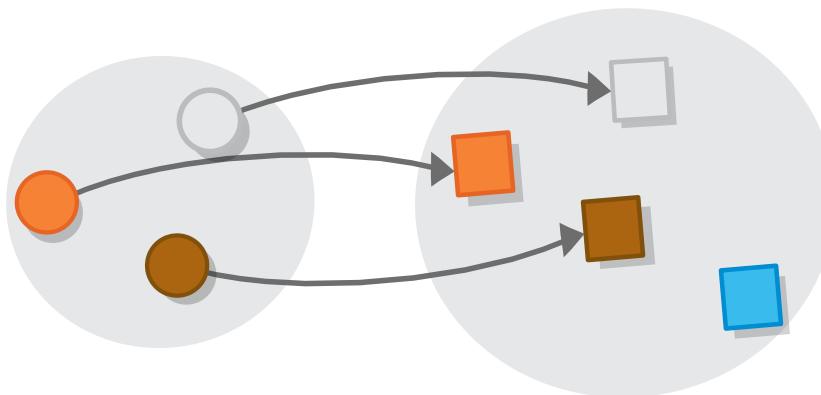


函子

现在让我们逐一解读这个定义的各个部分。

对象映射 (Object mapping)

在上面的定义中, 我们使用“映射”一词来避免误用“函数”这个词, 虽然它不是完全符合传统定义的函数。但在这种情况下, 将映射称为函数几乎不算错——如果忽略态射, 并将源范畴和目标范畴视为集合, 对象映射实际上就是一个普通的函数。



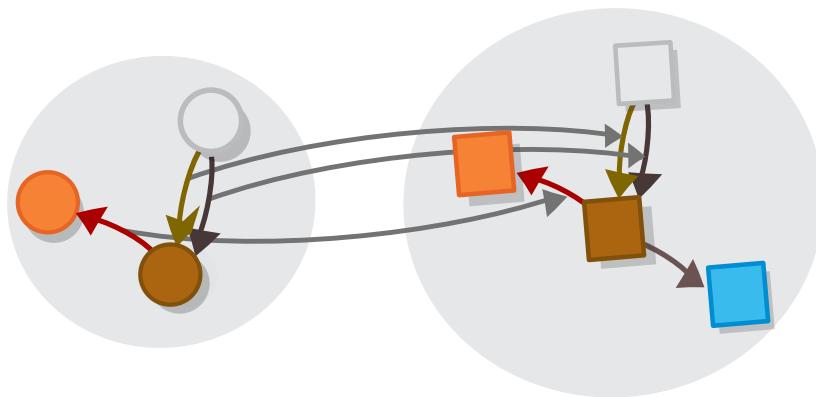
对象的函子

更正式的对象映射定义涉及到范畴的底层集合的概念: 给定范畴 A , A 的底层集合是一个集合, 它的元素是 A 的对象。利用这个概念, 我们说两个范畴之间的对象映射是它们底层集合之间的函数。函数的定义仍然相同:

函数是两个集合之间的关系，将一个集合的每个元素（称为函数的源集合）与另一个集合中的一个元素匹配（称为函数的目标集合）。

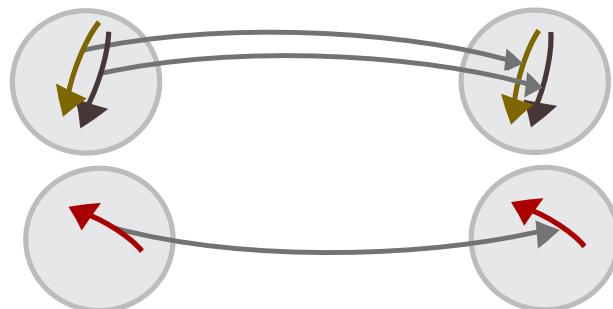
态射映射 (Morphism mapping)

构成函子的第二个映射是范畴态射之间的映射。这个映射也类似于函数，但附加了一个要求，即 A 中具有特定源和目标的态射必须映射到 B 中具有相应源和目标的态射，符合对象映射的定义。



态射的函子

更正式的态射映射定义涉及到同态集 (homomorphism set) 的概念：它是一个包含在给定范畴中两个对象之间所有态射的集合。利用这个概念，我们说两个范畴的态射映射由它们各自的同态集之间的一组函数构成。



态射的函子

(注意, 我们使用同态集和底层集合的概念, 从集合论“逃逸”到范畴论中, 并使用函数来定义一切。)

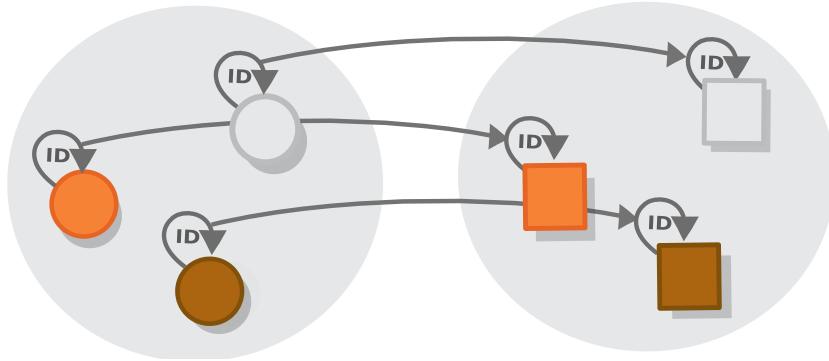
函子定律 (Functor laws)

到目前为止, 我们看到了构成函子的两个映射(一个是对象之间的映射, 另一个是态射之间的映射)。但并不是每一对这样的映射都构成函子。正如我们所说, 除了存在映射之外, 这些映射还必须保持源范畴的结构到目标范畴。为了理解这意味着什么, 我们回顾第2章对范畴的定义:

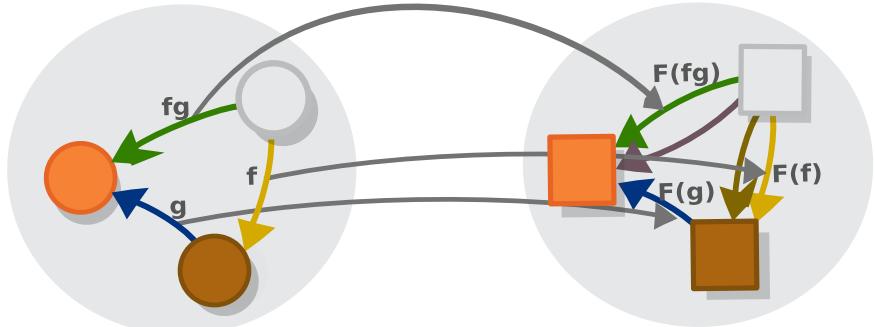
范畴是对象(可以看作点)和态射(箭头)组成的集合, 其中态射从一个对象到另一个对象, 并满足: 1. 每个对象必须具有恒等态射。2. 必须有一种方式将两个具有适当类型签名的态射组合成第三个态射, 并且这种组合是结合律的。

因此, 这一定义转化为以下两条函子定律:

1. 态射之间的函数应保持恒等性, 即所有恒等态射应映射到其他恒等态射。



2. 函子还应保持复合, 即对于任意两个态射 f 和 g , 在源范畴中对应于它们复合的态射 $F(g \circ f)$ 应该被映射到目标范畴中对应的复合态射 $F(g) \circ F(f)$ 上, 即 $F(g \circ f) = F(g) \circ F(f)$ 。



函子定律 - 复合性

这两条定律完成了对函子的定义——这是一个简单但非常强大的概念，我们接下来会看到它的强大之处。

日常语言中的函子 (Functors in everyday language)

在日常生活中有一个常见的说法（本书中也经常使用这种说法），如下所示：

如果 a 类似于 Fa , 那么 b 也类似于 Fb 。

或者说“ a 与 Fa 之间有某种关系，类似于 b 与 Fb 之间的关系”，例如“如果学校像公司，那么老师就像老板”。

这种说法实际上是在用日常语言描述函子的一种方式：我们的意思是，在学校和老师之间有某种联系（用范畴论的术语来说是“态射”），类似于公司和老板之间的联系，即存在某种保持结构的映射，将与学校相关的事物范畴映射到与工作相关的事物范畴，将学校 (a) 映射到公司 (Fa)，将老师 (b) 映射到老板 (Fb)，并且学校与老师之间的关系 ($a \rightarrow b$) 被映射为公司与老板之间的关系 ($Fa \rightarrow Fb$)。

图表是函子 (Diagrams are functors)

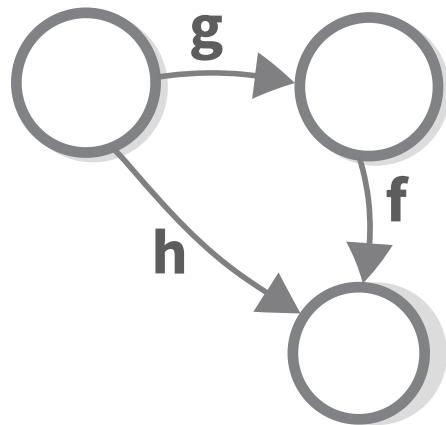
“符号是一种通过了解它，我们可以了解更多东西的存在。”——查尔斯·桑德斯·皮尔斯 (Charles Sanders Peirce)

我们将从一个非常元的函子例子开始——本书中的图表/插图。

你可能已经注意到，图表在范畴论中扮演了特殊的角色——在其他学科中，图表的功能仅仅是辅助性的，即它们只是展示已经通过其他方式定义的内容，而在这里，图表本身就是定义。

例如，在第1章中我们给出了以下函数复合的定义。

两个函数 f 和 g 的复合是一个第三个函数 h ，其定义方式使得该图表可交换。



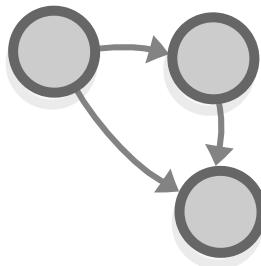
函数复合 - 通用定义

我们都看到了通过图表定义事物的好处，而不是写下冗长的定义，比如：

“假设你有三个对象 $a \cdot b$ 和 c ，以及两个态射 $f: b \rightarrow c$ 和 $g: a \rightarrow b \dots$ ”

然而，用图表定义事物会带来一个问题——数学中的定义应该是正式的，所以如果我们想使用图表作为定义，我们必须首先形式化图表本身的定义。

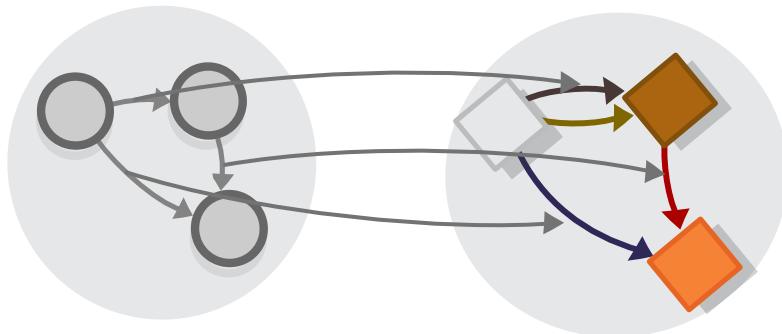
那么我们该如何做呢？一个关键的观察是，图表看起来像有限范畴，例如，上述定义与范畴 3 看起来相似。



有限范畴 3

然而,这只是故事的一部分,因为有限范畴只是结构,而图表是符号。它们是“通过了解它,我们可以了解更多东西”,正如皮尔斯著名的说法(或者用翁贝托·艾柯 (Umberto Eco) 的话来说,“……可以用来撒谎的东西”)。

因此,除了编码图表结构的有限范畴外,图表的定义还必须包括一种在其他上下文中“解释”这个范畴的方式,即它们必须包括函子。



图表作为函子

这就是函子概念让我们形式化图表概念的方式:

一个图表由

一个有限范畴(称为索引范畴)和从它到其他范畴的函子组成。

如果你了解符号学,您可以将函子的源范畴和目标范畴视为能指和所指。

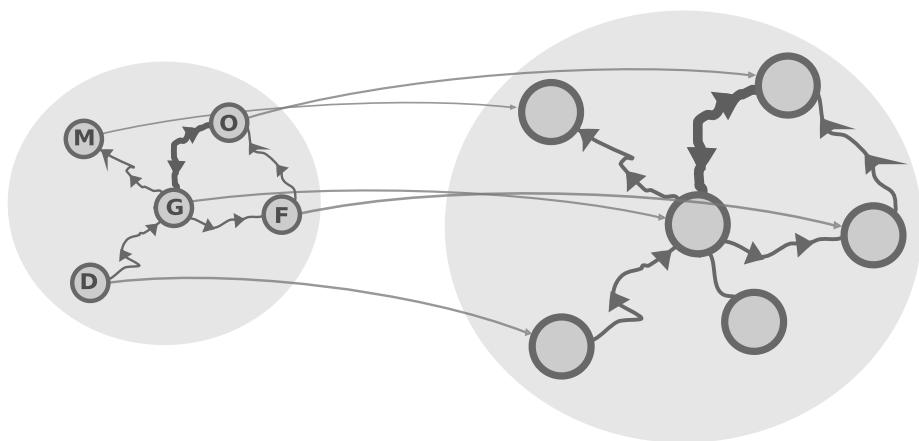
由此你可以看出，函子在范畴论中扮演着非常重要的角色。正因为如此，范畴论中的图表可以形式化指定，即它们本身就是范畴对象。

你甚至可以说它们是范畴对象的卓越代表(TODO: 删除最后这个笑话)。

地图是函子 (Maps are functors)

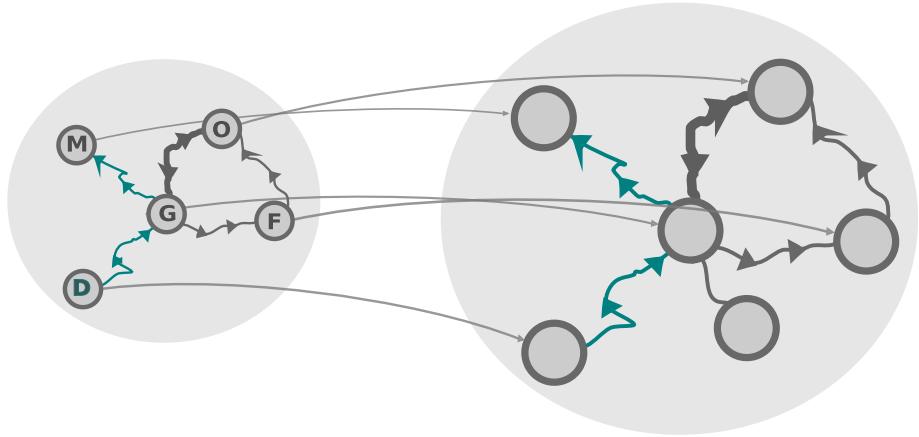
“地图不是它所代表的领土，但如果是正确的，它具有与领土相似的结构，这使它具有实用性。”——阿尔弗雷德·科日布斯基 (Alfred Korzybski)

函子有时被称为“映射”是有原因的——地图，就像其他所有图表一样，实际上是函子。如果我们将包含城市和连接城市的道路的空间视为一个范畴，其中城市是对象，道路是态射，那么一张路线图可以被视为代表该空间某一区域的范畴，以及将该地图中的对象映射到现实世界对象的函子。



城市路径的预序与地图

在地图中，通常不会显示复合得到的态射，但我们经常使用它们——它们被称为路线。保持复合性的定律告诉我们，我们在地图上创建的每条路线都对应现实世界中的一条路线。



城市路径的预序与地图 - 路线

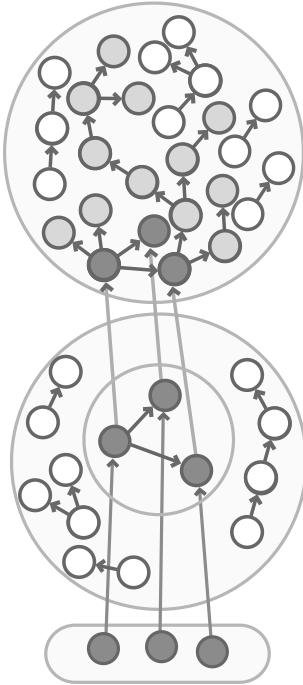
注意,为了成为一个函子,地图不必列出所有现实中存在的道路和所有旅行选项(“地图不是领土”),唯一的要求是它列出的道路必须是真实存在的——这是所有多对一关系(即函数)共有的特性。

人类感知具有函子性质 (Human perception is functorial)

我们看到,除了是一个范畴论概念之外,函子还与许多研究人类心智的学科相关,如逻辑、语言学、符号学等。为什么会这样呢?我最近写了一篇[关于使用逻辑建模现实生活思维的博客文章](#),其中探讨了函子(以及“映射”)的“非凡有效性”,我认为人类的感知和思维是具有函子性质的。

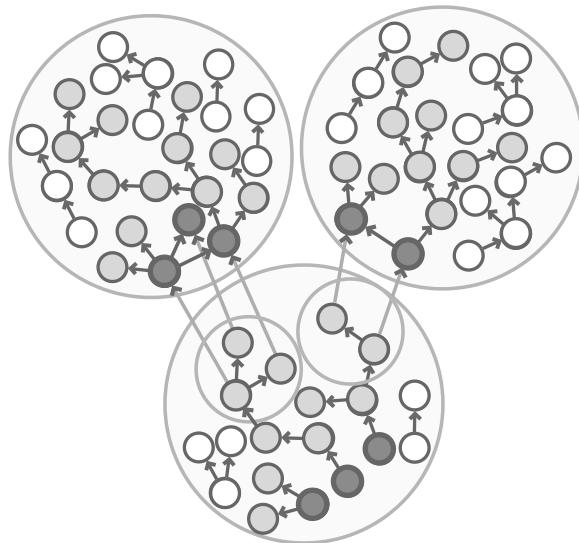
我的观点是,为了感知我们周围的世界,我们通过一系列函子,从更原始的“低层次”心理模型转向更抽象的“高层次”模型。

我们可以说,感知始于原始的感官数据。然后,通过一个函子,我们转向一个包含基本世界模型的范畴(告诉我们我们在空间中的位置、我们看到的物体数量等)。接着,我们将这个模型与另一个更抽象的模型连接起来,该模型为我们提供了所处情况的更高层次视图,依此类推。



感知具有函子性质

你可以将这视为从简单到抽象的进程，从具有较少态射的范畴进展到具有更多态射的范畴——我们从没有任何联系的感官数据组成的范畴开始，接着进入另一个范畴，其中一些数据片段之间有了联系。然后，我们将这种结构转移到具有更多联系的另一个范畴。



感知具有函子性质

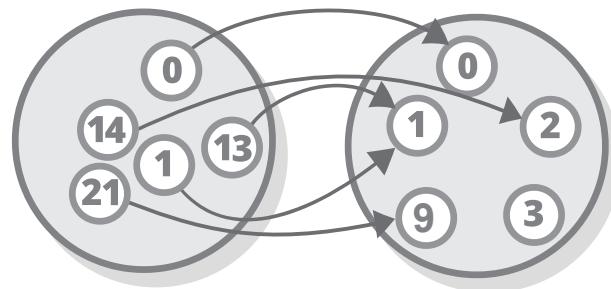
当然, 这只是一个猜想, 但当我们看到函子对于我们之前讨论的数学结构有多重要时, 或许可以证明它有一定的依据。

单子中的函子 (Functors in monoids)

在这个小插曲之后, 让我们回到我们通常的操作方式:

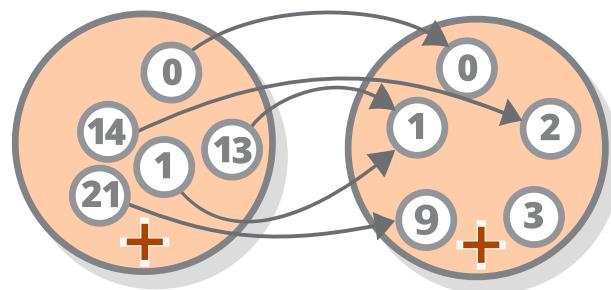
嘿, 你知道在群论中有一个很酷的东西叫做群同态(当我们谈论单子时, 它被称为单子同态)——它是一个在群的底层集合之间保持群运算的函数。

例如, 如果现在是 00:00(或中午 12 点), 那么 n 小时后会是什么时间? 这个问题的答案可以通过一个以整数集为源和目标的函数来表达。



群同态作为函数

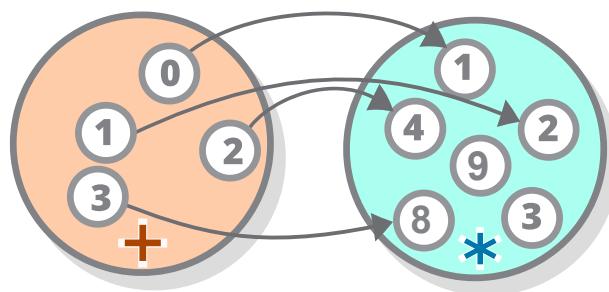
这个函数很有趣——它保持(模)加法运算: 如果 13 小时后是 1 点, 而 14 小时后是 2 点, 那么 $(13 + 14)$ 小时后是 $(1 + 2)$ 点。



群同态

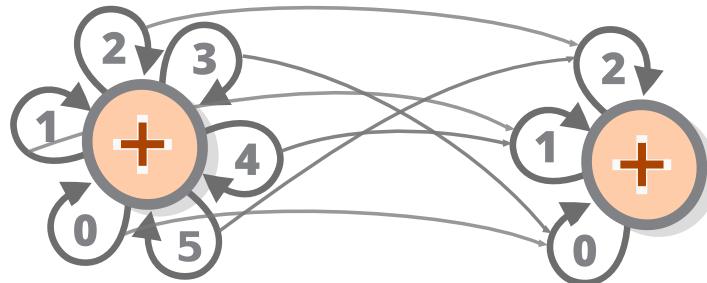
或者正式地说，如果我们称这个函数为 F ，那么我们有以下等式： $F(a + b) = F(a) + F(b)$ （其中右边的 $+$ 表示模加法）。由于这个等式成立， F 函数是一个群同态，它将整数加法群映射到模 11 加法群（你可以将 11 替换为任何其他数）。

群不必如此相似才能在它们之间存在同态。举个例子，将任何数 n 映射为 2 的指数 $n \rightarrow 2^n$ ，这个函数给出了整数加法群和整数乘法群之间的群同态，即 $F(a + b) = F(a) \times F(b)$ 。



不同群之间的群同态

哦对了，我们在讨论什么来着？对，群同态是函子。要理解为什么，我们切换到范畴论视角，并重新审视我们第一个例子（为了简化图表，我们使用模 2 代替模 11）。



群同态作为函子

看来，当我们把群/单子视为单对象范畴时，群/单子同态实际上就是这些范畴之间的函子。让我们看看是不是这样。

对象映射 (Object mapping)

当群/单子被视为范畴时，它们只有一个对象，所以在任何两个群/单子之间只有一个可能的对象映射——将源群的唯一对象映射到目标群的对象（图中未绘制）。

态射映射 (Morphism mapping)

因此，对于群同态来说，态射映射是唯一相关的组成部分。在范畴论视角中，群的对象（如 1、2、3 等）对应于态射（如 +1、+2、+3 等），因此态射映射就是群对象之间的映射，如图所示。

函子定律 (Functor laws)

第一个函子定律是平凡的，它只是说源群的唯一恒等对象（对应于其唯一对象的恒等态射）应该映射到目标群的唯一恒等对象。

如果我们记住，群的结合运算（组合两个对象）对应于将群视为范畴时的函数复合，那么我们会意识到群同态方程 $F(a + b) = F(a) \times F(b)$ 只是第二个函子定律 $F(g \circ f) = F(g) \circ F(f)$ 的一种表述。

许多代数运算都满足这个方程，例如群同态的函子定律 $n \rightarrow 2^n$ 就是著名的代数规则 $g^a g^b = g^{a+b}$ 。

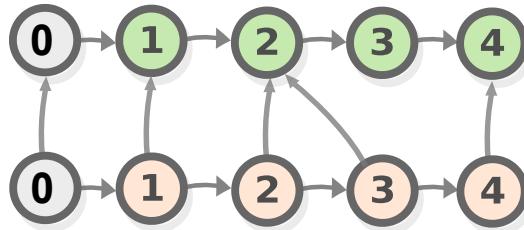
任务：尽管很简单，但我们没有证明第一个函子定律（关于保持恒等性的定律）总是成立。有趣的是，对于群/单子来说，它实际上是从第二个定律推导出来的。试着证明这一点。首先从恒等函数的定义开始。

序中的函子 (Functors in orders)

现在让我们谈论一个与函子完全无关的概念，开个玩笑（嘿，冷笑话总比没有笑话好，对吧？）。在序的理论中，我们有序之间的函数（这并不奇怪，因为序就像单子/群一样，基于集合），其中一个非常有趣的函数类型，具有在微积分和分析中的应用，就是单调函数（也叫做单调映射）。这是两个序之间的一个函数，它保持源序对象的顺序。

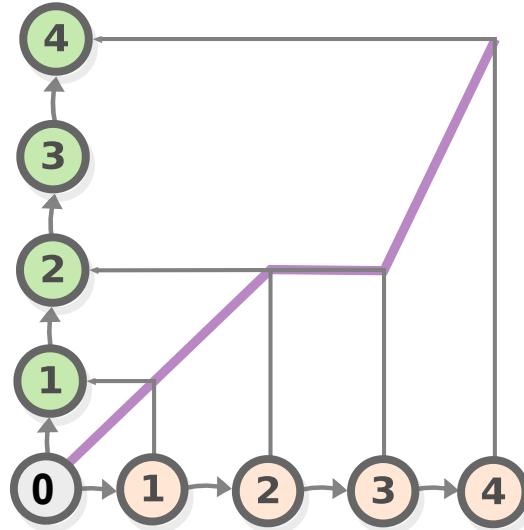
序，在目标序中也保持对象的顺序。所以当对于源序中的每一个 a 和 b ，如果 $a \leq b$ ，那么 $F(a) \leq F(b)$ ，此时函数 F 是单调的。

例如，将当前时间映射到某物体行驶距离的函数是单调的，因为行驶距离随着时间增加（或保持不变）而增加。



单调函数

如果我们在折线图上绘制这个或任何其他单调函数，我们会看到它的趋势只有一个方向（即只向上或只向下）。



单调函数，折线图表示

现在我们要证明单调函数也是函子，准备好了吗？

对象映射 (Object mapping)

像在范畴中一样,序的对象映射是其底层集合之间的函数。

态射映射 (Morphism mapping)

与单子不同,函子的对象映射部分是平凡的。这里正好相反:态射映射是平凡的——给定源序中的两个对象之间的态射,我们将该态射映射到目标序中对应的态射。单调函数尊重元素的顺序,确保后者态射的存在。

函子定律 (Functor laws)

不难看出,单调映射遵循第一个函子定律,因为恒等态射是唯一的,存在于每个对象与自身之间。

第二条定律 ($F(g \bullet f) = F(g) \bullet F(f)$) 也显然成立: $F(g \bullet f)$ 和 $F(g) \bullet F(f)$ 两个态射具有相同的类型签名。而在序中,具有给定类型签名的态射只能有一个,因此这两个态射必然是相等的。

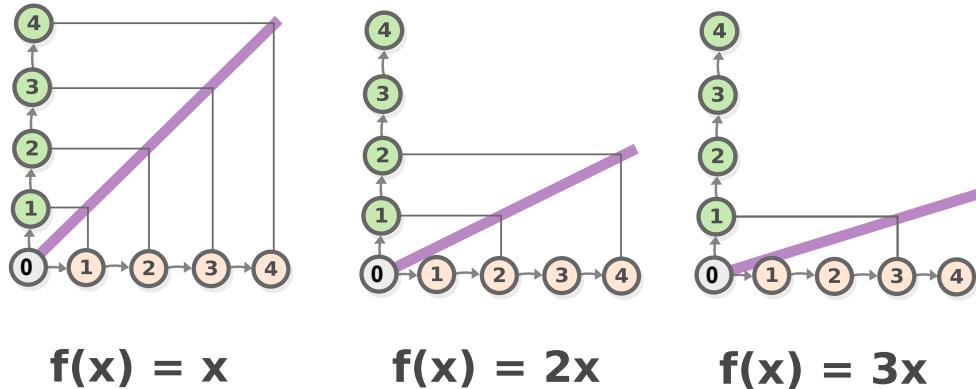
任务: 扩展证明。

线性函数 (Linear functions)

好了，足够的抽象内容了，让我们来谈谈“普通”函数——那些在数字之间的函数。

在微积分中，有一个线性函数（也称为“一次多项式”）的概念，通常定义为形如 $f(x) = ax$ 的函数，即那些只包含将自变量乘以某个常数（在示例中指定为 a ）的运算的函数。

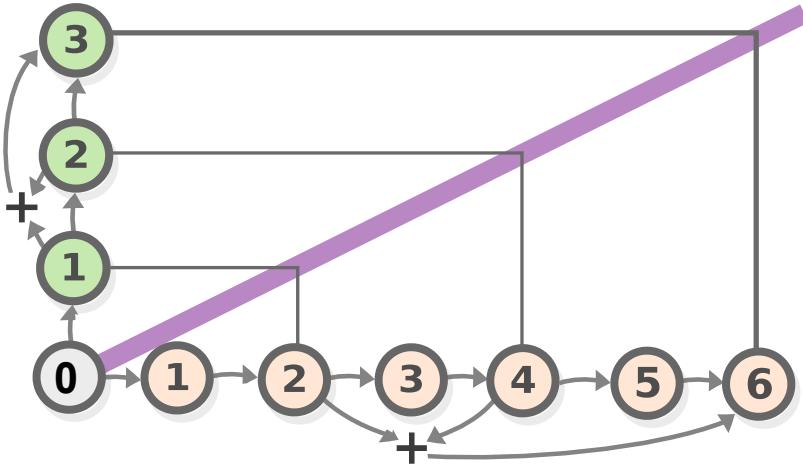
但如果我们开始绘制这些函数的图像，我们会意识到它们可以用另一种方式描述——它们的图像总是由直线组成。



线性函数

问题：为什么是这样？

这些函数的另一个有趣性质是，大多数函数保持加法，也就是说对于任意的 x 和 y ，我们有 $f(x) + f(y) = f(x + y)$ 。我们已经知道这个等式等价于函子的第二条定律。因此，线性函数实际上是加法下自然数群与其自身之间的函子。我们稍后会看到，它们是向量空间范畴中函子的一个例子。

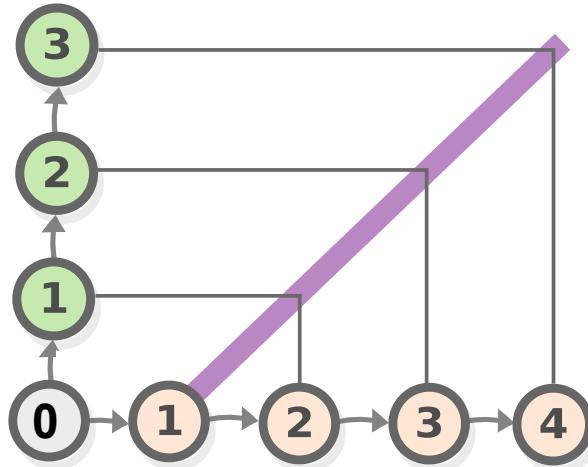


线性函数

问题： 我们用来定义线性函数的这两个公式是否完全等价？

如果我们将自然数视为一个序列，线性函数也是函子，因为所有用直线绘制的函数显然都是单调的。

但请注意，并非所有用直线绘制的函数都保持加法——形如 $f(x) = x * a + b$ 的函数，其中 b 非零，也是一条直线（并且也称为线性函数），但它们不保持加法。



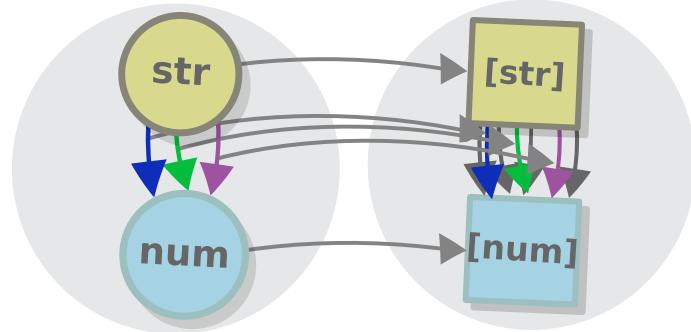
线性函数

对于这些函数，上述公式变为： $f(x) + b + f(y) + b = f(x + y) + b$ 。

编程中的函子——列表函子

(Functors in programming. The list functor)

编程语言中的类型构成了一个范畴，与该范畴相关的一些函子是程序员每天都会使用的，例如我们将使用的列表函子。列表函子是将简单（原始）类型和函数的领域映射到更复杂（泛型）类型和函数的领域的一个函子示例。



编程中的函子

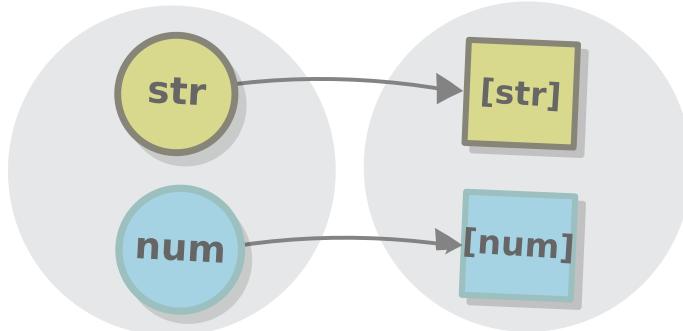
但让我们从基础开始：在编程上下文中定义函子的概念其实很简单，只需要根据第2章中的对照表（其中比较了范畴论和编程语言），并且（或许更重要的是）将我们公式中的字体从“现代”更改为“等宽字体”。

两个范畴之间的函子（我们称它们为_A 和 _B）由一个映射组成，它将_A 中的每个 对象 类型 映射到 _B 中的一个类型，并将_A 中每个 态射 函数 映射到 _B 中的一个函数，且映射方式保持范畴的结构。

比较这些定义，我们会意识到，数学家和程序员是两个非常不同的群体，但它们因都使用函子这一事实而联系在一起（并且它们都欣赏特殊的字体）。

类型映射 (Type mapping)

函子的第一个组成部分是将一种类型(我们称之为_A)转换为另一种类型(_B)的映射。所以它像一个函数,但在类型之间。几乎所有支持静态类型检查的编程语言都支持这种结构——它们称为泛型类型。泛型类型实际上就是一个将一种(具体)类型映射到另一种类型的函数(这就是为什么泛型类型有时被称为类型级函数)。

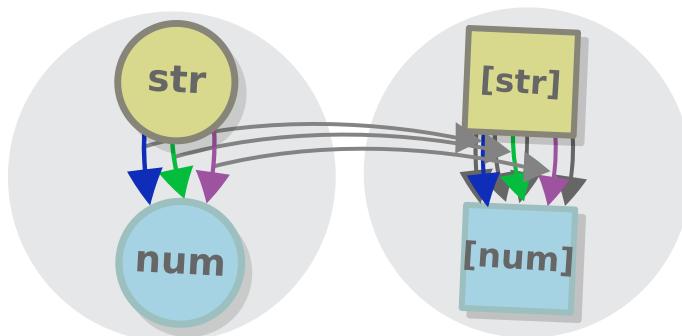


编程中的函子——类型映射

请注意,尽管图表看起来相似,但类型级函数与值级函数完全不同。一个从 `String` 到 `List<String>` 的值级函数(或使用数学风格的 Haskell/ML 风格的符号表示为 $string \rightarrow List\ string$)将 `String` 类型的值(例如 "foo")转换为 `List<String>` 类型的值。你甚至可以有(如我们稍后将看到的)一个签名名为 $a \rightarrow List\ a$ 的值级函数,它可以将任何值转换为一个包含该值的元素列表,但这与类型级函数 `List<A>` 不同,后者将类型 a 转换为类型 `List\ a`(例如,将类型 `string` 转换为类型 `List\ string`,将 `number` 转换为 `List\ number` 等)。

函数映射 (Function mapping)

因此,函子的类型映射只是编程语言中的泛型类型(我们也可以有两个泛型类型之间的函子,但我们稍后会讨论这些)。那么函数映射是什么呢——这是一个将任何操作简单类型的函数(如 $string \rightarrow number$)转换为它们更复杂的对应物之间的函数的映射,例如 $List\ string \rightarrow List\ number$ 。



编程中的函子——函数映射

在编程语言中，这种映射由一个名为 `map` 的高阶函数表示，其签名为（使用 Haskell 符号）， $(a \rightarrow b) \rightarrow (Fa \rightarrow Fb)$ ，其中 F 表示泛型类型。

请注意，尽管任何具有这种类型签名的可能函数（并且遵守函子定律）都会产生一个函子，但并非所有这样的函子都是有用的。通常，对于给定的泛型类型，只有一个函数是有意义的，这就是为什么我们谈论列表函子，并且看到 `map` 直接定义在泛型数据类型中，作为一个方法。

在列表和类似结构的情况下，`map` 的有用实现是将原始（简单）函数应用于列表的所有元素。

```
class Array<A> {
    map (f: A → B): Array<B> {
        let result = [];
        for (obj of this) {
            result.push(f(obj));
        }
        return result;
    }
}
```

函子定律 (Functor laws)

除了通过在更复杂的上下文中引入所有简单类型的标准函数来促进代码复用外，`map` 使我们能够以可预测的方式工作，这要归功于函子定律，在编程上下文中，它们看起来如下。

身份定律：

`a.map(a => a) == a`

组合定律：

`a.map(f).map(g) == a.map((a) => g(f(a)))`

任务：使用示例来验证这些定律是否被遵守。

函子的用途 (What are functors for)

现在，我们已经看到了这么多函子的例子，终于可以尝试回答这个价值百万美元的问题了，即函子到底是干什么用的，为什么它们有用？（通常也被表述为“你为什么要浪费你/我的时间在这些（抽象的）废话上？”）

我们已经看到了地图是函子，并且我们知道

地图是有用的，所以让我们从这里开始。

那么，为什么地图有用呢？显然，这与地图上的点和箭头对应于你所在地方的城市和道路这一事实有关，即因为它实际上是一个函子，但还有第二个方面——地图（或至少那些有用的地图）比它们所代表的实际事物更简单。例如，路线图之所以有用，是因为它们比它们所代表的区域小，因此查找两个地方之间的路线时，使用地图要比实际走遍所有路线简单得多。

在编程中使用函子的原因也是类似的——涉及简单类型如 `string`、`number`、`boolean` 等的函数是……简单的，至少与处理列表和其他泛型类型的函数相比是这样的。使用 `map` 函数允许我们在不需要考虑这些复杂类型的情况下操作它们，并从操作简单值的函数中派生出转换这些复杂类型的函数。换句话说，函子是一种抽象的手段。

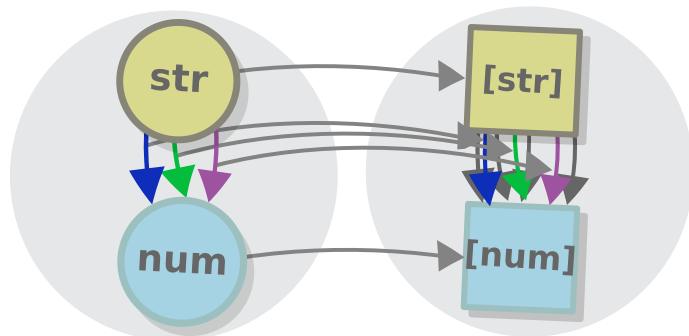
当然，并非地图上的所有路线和泛型数据类型之间的所有函数都可以仅通过它们包含的类型之间的函数派生出来。这通常适用于许多“有用的”函子：因为它们的源范畴比目标范畴“简单”，所以目标范畴中的一些态射在源范畴中没有对应物。即，简化模型不可避免地导致失去一些能力。这是“地图不是领土”原则的一个结果（或在编程上下文中，如 Joel Spolsky 所说的，“每个抽象都是有漏洞的抽象”）。

指向函子 (Pointed functors)

现在，在结束之前，我们将回顾一个在编程中特别有用的与函子相关
的概念——指向自函子(pointed endofunctors)。

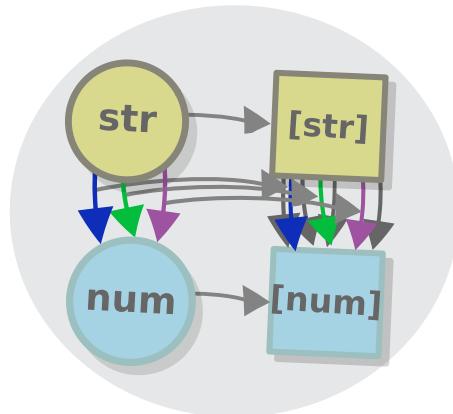
自函子 (Endofunctors)

要理解什么是指向自函子，我们首先要理解什么是自函子，我们在上一节中已经看到了它们的一些例子。让我解释一下：从那里的图看起来，我们可能会误以为不同的类型家族属于不同的范畴。



编程中的函子

但事实并非如此——来自给定编程语言的所有类型家族实际上属于同
一个范畴——**类型范畴**。

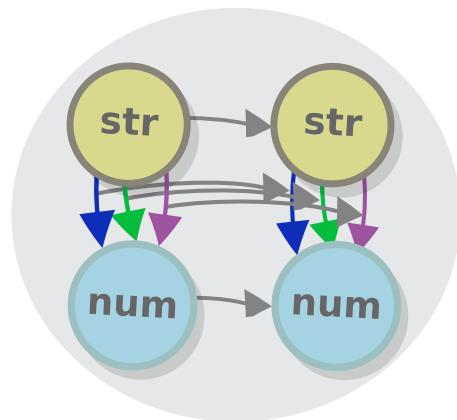


编程中的自函子

等等,这是允许的吗?是的,这正是我们所说的*自函子*,即那些源范畴和目标范畴相同的函子。

恒等函子 (The identity functor)

那么,自函子的一些例子是什么呢?我想集中讨论一个你可能会觉得熟悉的例子——它是每个范畴的*恒等函子*,它将每个对象和态射映射到其自身。

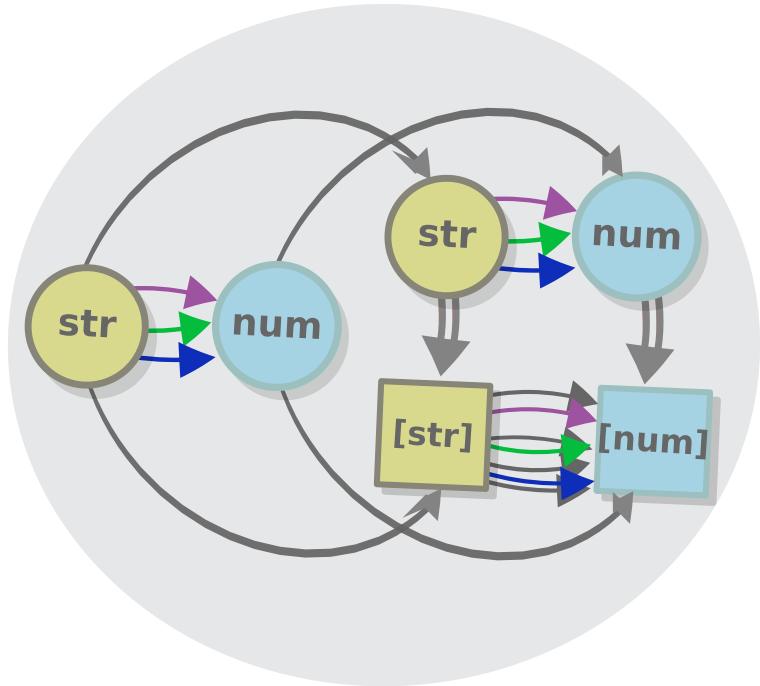


恒等函子

这可能会让你觉得熟悉,因为恒等函子类似于恒等态射——它允许我们讨论与值相关的内容,而不实际涉及值。

指向函子 (Pointed functors)

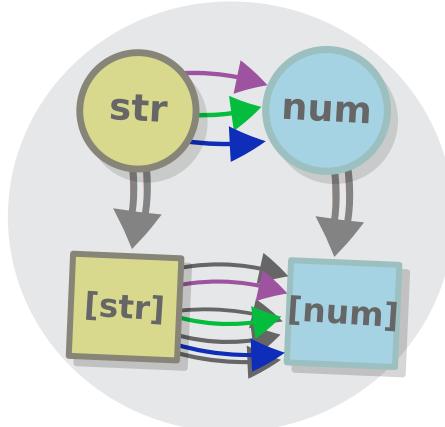
最后,恒等函子以及所有可以自然变换到恒等函子的函子被称为*指向函子*(即,如果存在从恒等函子到某个函子的态射,则称该函子为指向函子)。正如我们将很快看到的,列表函子是一个指向函子。



指向函子

我们还没有讨论什么是一个函子自然变换到另一个函子(尽管上面的交换图可以给你一些提示)。这是一个复杂的概念,我们将在下一章中详细探讨。

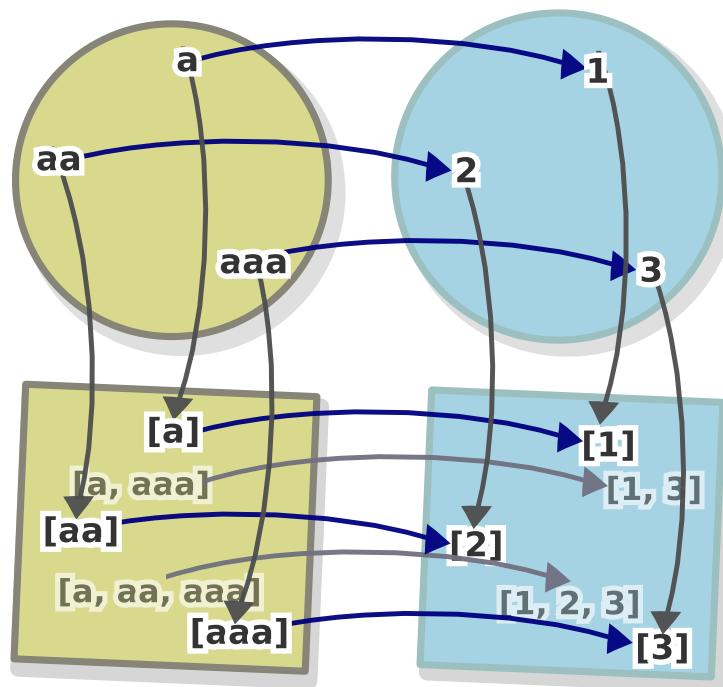
然而,如果我们仅关注编程语言中的类型范畴,那么自然变换只是一个函数,它将我们称为“简单类型”的每个值转换为函子的泛型类型的一个值,即 $a \rightarrow Fa$,并且这种方式使得这个图表交换。



集合范畴中的指向函子

让这个图表交换的条件是什么？这意味着，当你有两个等效的路径从左上角到右下角时，即应用任何两个类型之间的函数 ($a \rightarrow b$)，然后是提升函数 ($b \rightarrow Fb$)，等价于首先应用提升函数 ($a \rightarrow Fa$)，然后是原始函数的映射版本 ($Fa \rightarrow Fb$)。

列表函子是指向的，因为列表函子存在这样的函数——它是 $a \rightarrow [a]$ ，将每个值放入一个“单例”列表中。因此，对于简单类型之间的每个函数，例如 $length: string \rightarrow number$ ，我们有一个像这样的正方形。



集合范畴中的指向函子

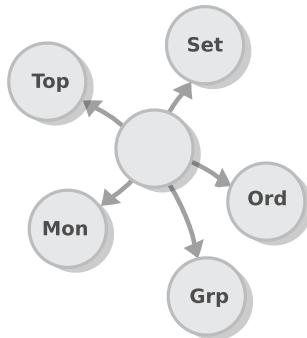
该方形的交换可以通过以下等式表示：

```
[a].map(f) = [f(a)]
```

顺便说一句，尽管它现在看起来可能并不像这样，但这个交换方形可能是范畴论中存在的最重要的图之一，仅次于函数组合的三角形。

小范畴的范畴 (The category of small categories)

哈哈,这次我抓住你了(至少我希望如此)——你可能认为我不会在本章中引入另一个范畴,但这正是我现在要做的事情。(再次惊喜)新引入的范畴不会是函子的范畴(别担心,我们会在下一章介绍它)。相反,我们将研究(小)范畴的范畴,它的对象是我们迄今为止看到的所有范畴,而它的态射是这些范畴之间的函子,比如 Set ——集合范畴, Mon ——幺半群范畴, Ord ——序范畴等。



范畴的范畴

我们还没有提到函子是可以组合的(并且是以关联的方式),但由于函子只是一些函数,毫不奇怪它是可以组合的。

任务: 通过函子的定义,看看它们是如何组合的。

问题: 小范畴范畴的始对象和终对象是什么?

层层范畴 (Categories all the way down)

范畴论的递归性质有时会让我们感到困惑:我们一开始说范畴是由对象和态射组成的,但现在我们又说范畴之间存在态射(函子)。不仅如此,还有一个范畴,其中对象本身是范畴。这是否意味着范畴是……范畴的一个例子?从直觉上讲,这听起来有点奇怪(例如,饼干不包含其他

饼干，房子也不会用房子作为建材），但事实确实如此。例如，每个幺半群都是只有一个对象的范畴，但同时，幺半群可以被看作属于一个范畴——幺半群范畴，它们通过幺半群同态相互连接。我们还可以举例群范畴，它包含了幺半群范畴作为子范畴，因为所有幺半群都是群，等等。

范畴论对一切进行分类，因此，从范畴论的角度来看，整个数学都是层层范畴。你是否将给定的范畴视为一个宇宙或一个点，完全取决于上下文。范畴论是一种抽象理论。即，它并不寻求代表实际的状态，而是提供一种可以用来表达许多不同想法的语言。