Ideograph：用于表达和操纵结构化数据的语言

StephenMell OsbertBastani SteveZdancewic宾夕法尼亚大学
宾夕法尼亚大学 宾夕法尼亚大学
sm1@cis.upenn.edu obastani@seas.upenn.edu stevez@seas.upenn.edu

我们介绍了 Ideograph，这是一种用于表达和操纵结构化数据的语言。其类型描述了各种结构的种类，例如自然数、列表、多重集、二叉树、带变量绑定的语法树、有向多重图和关系数据库。某一类型的完全规范化项与该结构的成员一一对应，类似于丘奇编码。此外，在这些结构上可定义的操作保证尊重这些结构的等价关系。在本文中，我们给出了 Ideograph 的非多态子集的语法和语义，并展示了如何利用它表示和操纵若干有趣的结构。

1 引言

结构化数据无处不在：列表、树、图、关系数据库和语法树只是支撑计算机科学的众多结构中的一小部分。我们经常希望以既遵循其结构、又利用其结构的方式对这类对象执行操作。例如，我们可能希望聚合多重集的元素。我们可以将多重集表示为列表，并像处理列表一样对其进行折叠，但这无法保证结果与元素顺序无关。或者，我们可能希望操作程序的语法树。我们可以将变量表示为名称或 de Bruijn 指标 [7], ，但无论采用哪种表示方式，都必须证明表示上的操作遵循绑定结构。实践中还经常会遇到其他类似的情况。

然而，真正用于定义这类结构的形式体系出奇地少，更不用说定义在这些结构上遵循不变量的操作了。关系数据库模式定义了记录的多重集，并带有某些额外结构（其中最突出的是表之间的外键约束）。大多数广泛使用的编程语言，如 C、Java 和 Python，以及数据交换格式，如 Google Protocol Buffers，都只有有限的类型系统，至多支持积类型、和类型和函数类型，却不支持定义带有变量绑定的多重集或语法树所需的图结构和约束。依赖类型语言，如 Coq，能够施加复杂的约束，但即使是简单的数据结构，例如带有绑定结构的语法树，在实践中也已被证明难以处理 [4]。

因此，我们不得不采用临时解决方案。对于在多重集上进行聚合，我们可以单独证明函数与顺序无关，从而确认它确实是作用于多重集而非列表的函数。对于操作语法树，我们可以单独证明替换操作能够避免捕获。然而，对于每一种新的结构和操作，都必须重复进行这样的证明。我们希望有一种通用的形式体系，用于表示和操作一大类丰富的结构。

图是编码多种数据的常见形式体系，但并不能涵盖所有情况。例如，二叉树是“图”，但它们具有更丰富的结构：包含两种不同种类的节点（“分支节点”和“叶节点”）以及两种不同种类的边（“左子节点”和“右

C. Grabmayer（编）：TERMGRAPH 2022 会后论文集，EPTCS 377，2023 年，第 65–84 页，doi:10.4204/EPTCS.377.4

© S. Mell、O. Bastani 和 S. Zdancewic
本作品采用知识共享署名许可协议授权。

子节点”），并且还要求：(1) 每个分支节点都有一个左子节点和一个右子节点；(2) 除根节点外，每个节点都有一个父节点。图的形式体系也无法描述操作：给定一棵叶节点本身标记有其他二叉树的二叉树，我们可能想把这棵由树组成的树折叠成一棵树。图是一个很好的起点，但就其本身而言，还不足以成为精确描述这些结构和操作的形式体系。

多态λ演算中的丘奇编码能够精确表达许多这类结构，并提供一种自然的、尊重结构的操纵方式。例如，类型 $\forall X.\,(X \to X \to X) \to (Y \to X) \to X$ 恰好编码了叶节点以 Y 中的元素为标签的二叉树。（大致来说，这两个参数分别对应二叉树中的两种节点： $X \to X \to X$ 对应分支节点，它们有两个子树子级和一个父节点； $Y \to X$ 对应叶节点，它们有一个 Y 子级和一个父节点。）此外，结构的丘奇编码本身就是函数，对应于广义折叠操作：要使用一个项，你需要为每个构造器提供一个函数，并将项中每次出现的构造器替换为相应的函数调用。这使得我们能够以尊重结构的方式操纵项。然而，标准的丘奇编码针对的是异构项代数，而多重集、关系数据库模式以及带有变量绑定的语法树并不是项代数。最后，这些编码并非唯一，例如，$\forall X.\,(Y \to X) \to (X \to X \to X) \to X$ 也能编码二叉树。随着编码结构变得越来越复杂，等价编码的数量可能会呈组合式增长。

在这项工作中，我们结合了这两种方法各自的优势，构建了一种名为 Ideograph 的语言，其中项和类型都采用图结构。借助演算系统，我们能够精确地编码多种结构，并定义作用于这些结构、保持其结构的操作。由于项是图而非树，我们能够表示更丰富的一系列结构。由于类型采用图结构，我们能够消除许多冗余的结构编码。

我们首先通过示例介绍项（第 2.1 节）、操作语义（第 2.2 节）、类型（第 2.3 节）和良构条件（第 2.4 节），随后给出形式体系（第2.5节）。接着，我们展示如何在该语言中表示几种数据结构（第 3 节），并演示如何操作这些结构（第 4 节）。最后，我们讨论相关工作（第 5 节）和未来工作（第 6 节）。为使内容清晰简洁，本文省略了多态性。

2 Ideograph

Ideograph 是一种表达和组合结构化图的方法。它的项之所以称为“结构化”，是因为边和节点具有不同的种类。节点具有“端口”，边通过这些端口连接节点。尽管我们引入了额外的构造来支持计算和多态性，其核心思想仍是将一个图的副本替换到另一个图中的某些节点上。由于语法和语义的形式化定义涉及许多环节，若缺乏背景知识便难以理解，因此我们先通过图 1 中的示例逐步讲解，这些示例展示了 Ideograph 的关键方面。形式体系将在第2.5节介绍。

2.1 项

Ideograph 中的项（下文简称为 意符）由一组框 （$\mathcal{B}$，例如图 1 中以圆角矩形表示）、节点 （$\mathcal{N}$，灰色圆形或矩形）和端口 （$\mathcal{P}$，小三角形

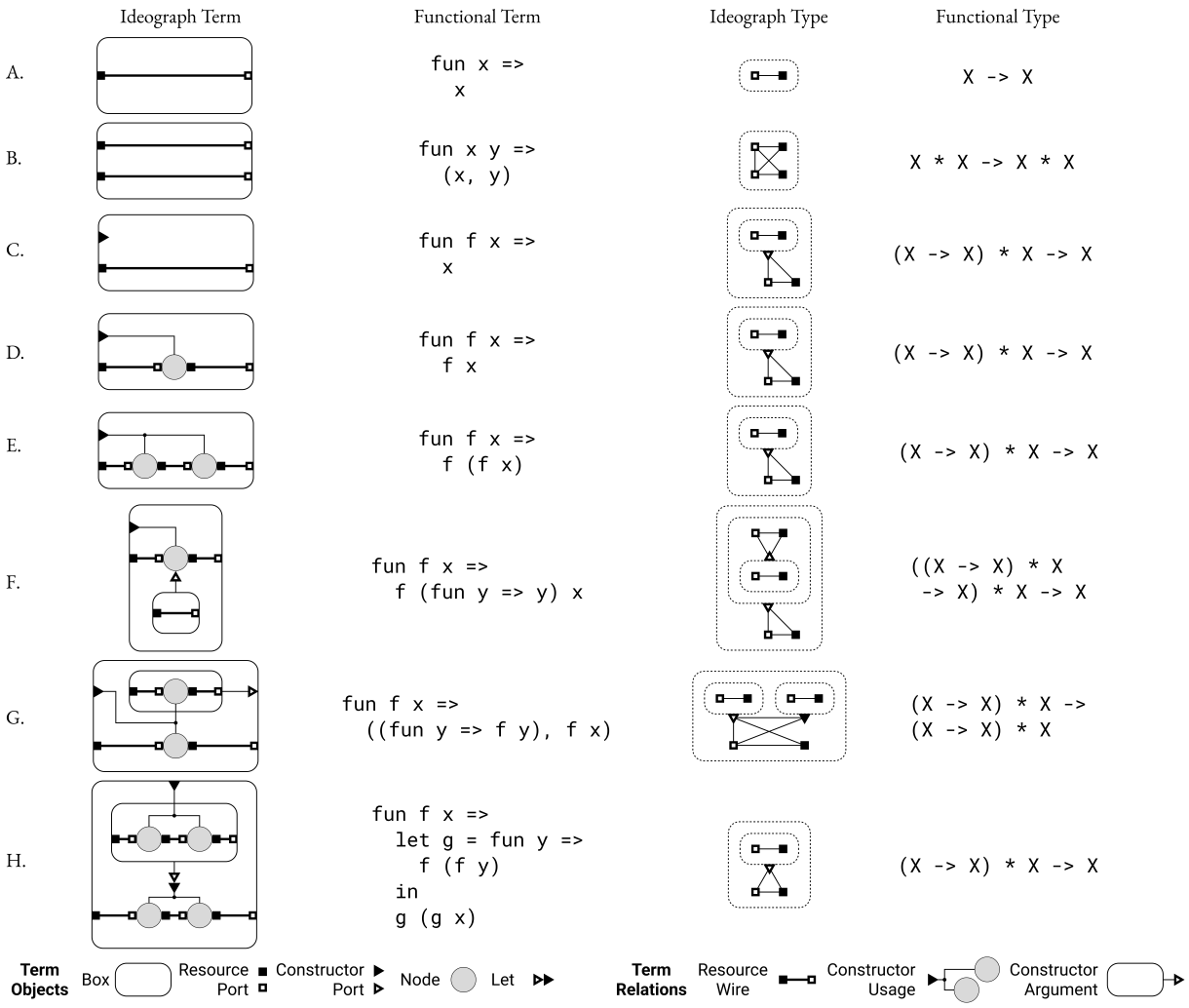


图 1：Ideograph 中的项及其类型，以及它们在通用函数式编程语言中的对应形式。尽管可能有多个 Ideograph 类型能够表示同一种函数类型，但这些示例应能提供恰当的直觉。更准确地说，Ideograph 的类型系统在 Girard [11], 的意义上是线性的；在这里，我们将标准函数类型 $X \to Y$译为 !($X \multimap Y$)。这样，函数类型参数便可重复使用，而其他参数则是线性的。不过，还有其他译法 [14], ，这里纯粹是出于说明目的而作此选择。为简化表述，我们在类型中只使用一种基本元素（X），因此给出的类型未必是其对应项的主类型。该形式体系允许使用其他基本元素为资源字段添加标记。

和正方形，空心或实心）组成，这些不同对象之间存在若干种关系。框、节点和端口位于其他框中（关系 R_R，以嵌套表示），框则构成一棵树。每个端口要么是接收端 （$\mathcal{P}_-$，空心），要么是提供端 （$\mathcal{P}_+$，实心）；它所对应的对象要么是资源 （$\mathcal{P}_R$，正方形），要么是构造器 （$\mathcal{P}_C$，三角形）。端口通常附着 于（R_A，以接触表示）节点或框上。随着下文示例的展开，我们将介绍这些对象之间的其他关系。图 4 和图7 展示了标注有这些对象及其关系的示意图。项还须满足良构条件，相关内容将在第2.4节讨论。完整的形式体系将在第2.5节介绍。

简单函数。考虑恒等函数（图 1 中的示例 A）。作为一个意符，它是一个盒，带有两个资源端口：一个资源提供端口（实心方块），类比于函数类比的输入 `x`；另一个是资源接收端口（空心方块），类比于函数输出。由于恒等函数将其输入作为输出返回，因此两个资源端口之间有一条连线，这由资源提供端口与资源接收端口之间的双射资源布线关系(R_{WR}，以粗线表示)体现。

示例 B 稍微复杂一些。它有两个资源提供端口，分别对应两个参数 `x` 和 `y`；还有两个资源接收端口，分别对应两个输出，即元组的左侧和右侧。两条连线表示哪个输入会作为哪个输出返回。由于资源布线关系是双射，因此不存在此类型的项可以类比于返回成对的 `(x, x)` 或 `(y, y)`。这使得资源具有线性特征。

调用函数。示例 C 与示例 A 相同，但增加了一个未使用的构造器提供端口（显示为实心三角形，有时简称为“构造器”），对应于类型为 `X -> X` 的未使用参数 `f`。这种未使用是允许的，因为构造器不像资源那样是线性的。

示例 D 和 E 更有趣，因为它们实际使用了构造器。节点类似于函数调用，因此在示例 D 中，我们有一个节点，对应于对 `f` 的一次调用。为了表示该节点由构造器提供端口构造，端口和节点通过构造器使用关系（R_{CU}）关联起来，该关系用一条细线表示，这条线可能会分支；分支结构没有实际意义，只是为了提高可读性。每个节点必须恰好与一个构造器关联，但构造器可以与任意数量的节点关联。在示例 E 中，该构造器用于构造两个节点，类似于对 `f` 的两次调用。

节点也可以像盒子一样关联端口。在示例 D 和 E 中，每个节点都有一个资源接收端口，对应于对 `f` 的调用的输入；还有一个资源提供端口，类似于输出。在示例 D 中，左侧的连线对应于将输入 `x` 传递给对 `f`的调用，右侧的连线则类似于返回 `f` 的输出。在示例 E 中，这些连线对应于将 `x` 传递给对 `f`的第一次调用，将其输出传递给对 `f`的第二次调用，最后返回其输出。请注意，节点和盒子对提供端和接收端的视角相反：调用函数（类似于节点）时，函数接收输入并提供输出；定义函数（类似于盒子）时，上下文提供输入并接收输出。

传递和返回函数。示例 F 与示例 D 类似，但 `f` 除了接受一个类型为 `X` 的参数外，还接受一个类型为 `X -> X`的参数。这类似于连接到节点的构造器接收端口（空心三角形）。传递恒等函数对应于嵌套盒子，即示例 A 的一个副本，它通过构造器参数关系 (R_{CA})连接到构造器接收端口。此关系是构造器接收端口与盒子（不包括顶层盒子）之间的双射。这是第一个具有非平凡盒子驻留结构的示例：有两个盒子，其中一个（图示为内层）是另一个（图示为外层）的子级。

示例 G 展示了函数如何返回函数：函数类型输出类似于构造器接收端口；如同示例 F 中那样，它必须连接到一个盒。请注意，构造器使用关系有时可以跨越框边界，这类似于函数的词法

作用域：在示例 G 中，与 `f` 对应的构造器提供端口在内部框中使用。形式上，构造器使用关系可以将节点关联到同一盒中的构造器提供端口，也可以关联到驻留树中更高层级的盒中的构造器提供端口。

let 绑定。 到目前为止，我们只见过值。要表示能够执行操作步骤的项，我们还需要 let 绑定（$\mathcal{D}$，由构造器接收端口与构造器提供端口的接触表示）。在示例 H 中，连接到接收端口的盒类似于 let 绑定体 `fun y => f (f y)`，而连接到提供端口的两个节点类似于 `g` 的两个实例。每个 let 绑定都必须恰好连接到一个构造器接收端口和一个构造器提供端口。每个端口都必须恰好关联一个盒、节点或 let 绑定。每个 let 绑定也有一个类型，将在第 2.3 节讨论。

2.2 操作语义

回想一下，Ideograph 的核心思想是用项替换其他项中的节点。let 绑定构造将一个盒（绑定的主体）连接到若干节点（绑定变量的出现位置）。Ideograph 的单一归约规则是用主体替换出现节点。（在多态性中，还有一种类型级 let 绑定，并且还有第二条用于在类型级进行替换的归约规则。）

请看图 2 (i)。let 绑定类似于 `g` 的定义，将两个节点串联起来，而每个节点都类似于对 `f` 的一次调用。接着，这个 let 绑定用于构造两个彼此串联的节点。步进会取出盒中的内容（蓝色），并将其副本放置在每个出现节点（橙色）处。图 2 (ii) 描绘了这一理解，但还需要稍作整理。每一对相邻的资源接收端口和提供端口都替换为一条连线。即使经过替换，`g` 主体中的 `f` 节点仍然是 `f` 节点，不过现在共有四个。最后，删除 let 绑定，得到图 2 (iii)。

尽管前面的示例抓住了将项替换为节点这一核心思想，但 let 绑定上没有构造器端口。图 3 则有。我们再次复制 let 绑定体 `h`，以替换其出现节点，然后擦除 let 绑定，并将每一对资源接收端口和提供端口替换为一条连线。关键的是，构造器接收端口和提供端口这一对端口会变成一个新的 let 绑定。这意味着该项可以继续步进。（事实上，图 3 (iii) 与图 2 (i) 是同一个项。）有关操作语义的形式化定义以及此示例的形式化，请参见第2.5节。

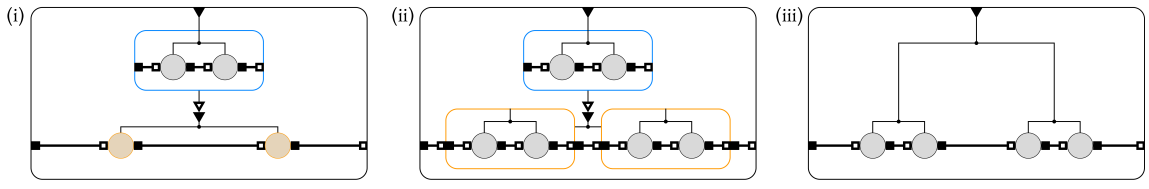


图 2：(i) 和 (iii) 展示了意符图及其函数类比物。(i) 通过内联 let 绑定，一步求值得到 (iii)。(ii) 不是一个项，而是展示了内联的过程。颜色表示类比关系。

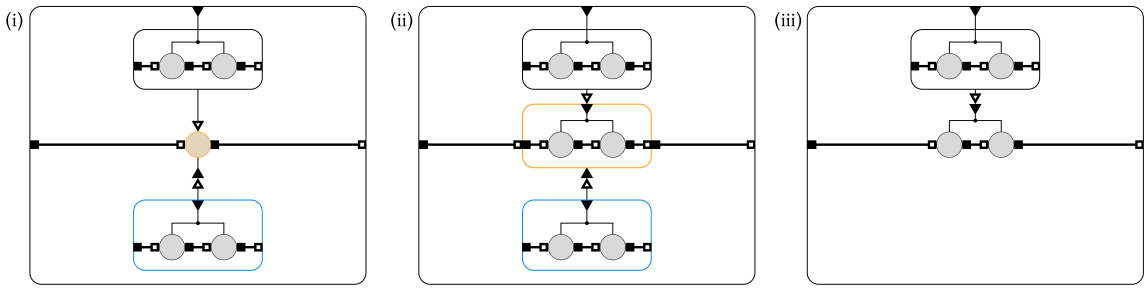


图 3：（i）和（iii）展示了意符图及其函数类比物。（i）通过内联 let 绑定，一步求值为（iii）。（ii）不是一个项，而是展示了如何进行内联。颜色表示类比关系。图7包含此示例的形式化标注版本。

2.3 类型与对应关系

按上述方式进行替换会带来一个难题：我们如何确定盒上的端口与节点上的端口之间的对应关系？图 2 假定节点上的左端口和右端口分别对应盒上的左端口和右端口。类型在 Ideograph 中的主要作用，就是明确这种对应关系。

现在我们定义一种类型，以及类型的分量（$\mathcal{I}$、$\mathcal{F}$，稍后定义）与项的分量（$\mathcal{B}$、$\mathcal{N}$、$\mathcal{P}$、$\mathcal{D}$）之间的对应关系。项中的端口在类型中对应的是字段（$\mathcal{F}$，其表示方式与端口相同），对应关系将每个端口映射到至多一个字段。在项中，端口附着于盒或节点；而在类型中，字段归属于接口（$\mathcal{I}$，以虚线圆角矩形表示），对应关系将每个盒和每个节点映射到至多一个接口。接口中的字段归属关系，以及接口的嵌套，由驻留关系（R_R，以包含关系表示）刻画，与项中的情况大致相同。对应关系必须保持一致：若某个盒或节点对应一个接口，则附着于该盒或节点的端口必须与该接口中的字段构成双射。与端口一样，字段可以是接收的（$\mathcal{F}_-$）或提供的（$\mathcal{F}_+$），并且可以属于构造器（$\mathcal{F}_C$）或资源（$\mathcal{F}_R$）。构造器端口和资源端口分别对应构造器字段和资源字段。附着于节点时，接收端口和提供端口分别对应接收字段和提供字段。然而，附着于盒时，情况正好相反：接收端口对应提供字段，提供端口对应接收字段。构造器字段与处于同一层级的接口构成双射（R_I，以接触表示）。对应关系还必须满足关于 R_I的一致性：若某个字段与一个接口相关联，则与该字段对应的端口只能连接到对应于该接口的盒（通过 R_{CA}）和节点（通过 R_{CU}）。最后，每个接口中都有一个字段之间的连通关系（R_C），将在第2.4节讨论。

示例。 在图示中，对应关系的表示有些隐含，是通过端口的定位来体现的（端口可以位于左、右、上、下、左上、右上、左下或右下）。在图 4 中，对应关系（a，A）通过将端口和字段都放置在其容器的左侧来表示，而（b，B）位于左上方，（c，C）位于右侧。由于 $\mathbf{Y}$与 B 相关联，且$\mathbf{y}$ 由 b构造，因此对应关系的定义要求

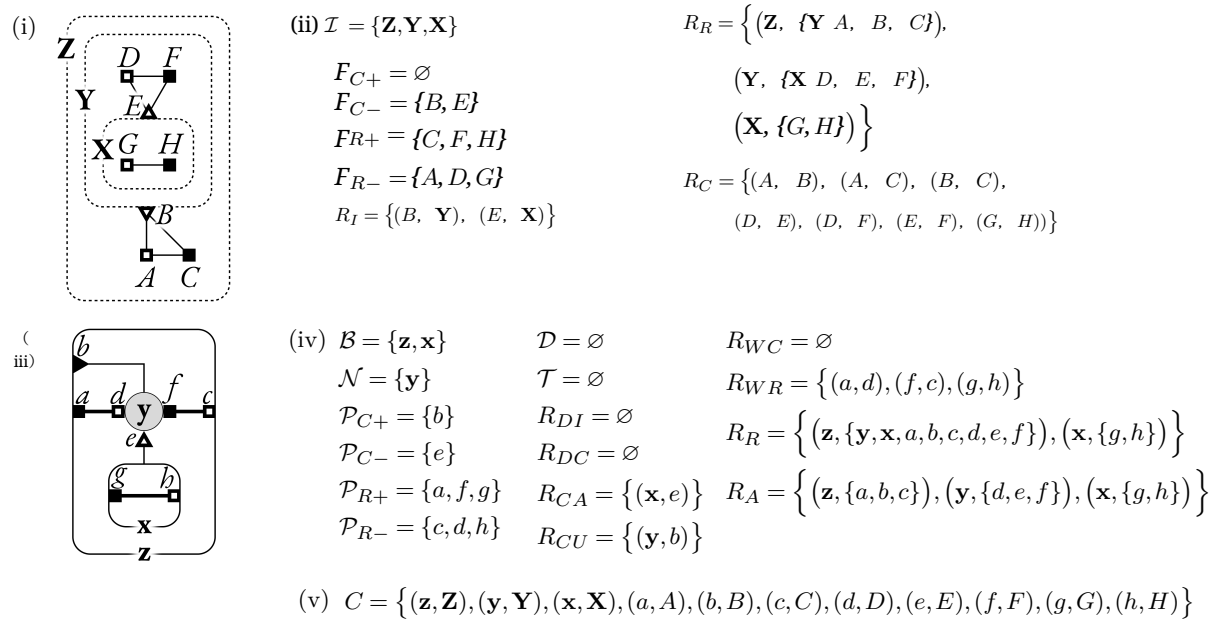


图 4：图 1 中类型 F 的注释图示 (i) 和形式化 (ii)。图 1 中项 F 的注释图示 (iii) 和形式化 (iv)。它们之间的对应关系 (v) 通过字段和端口的定位隐式表示。{(*a*、{*b*、 *c*})} 是 {(*a*、 *b*)，(*a*、 *c*)}的简写。所有形式体系分量的描述请参见图 8。

对应关系（$\mathbf{y}$，$\mathbf{Y}$）。将 d 放置在节点左侧、将 D 放置在接口左侧表示对应关系（d，D）；同样地，将（e，E）放置在底部、将（f，F）放置在右侧。对应关系（e，E）要求（$\mathbf{x}$，$\mathbf{X}$），随后，左侧和右侧的定位分别表示（g，G）和（h，H）。

项的内部类型。回顾一下如何在主体与 let 绑定的各个出现之间关联端口这一问题：解决方案是为每个 let 绑定指定一个类型，然后为主体和该 let 绑定的每个出现分别指定与该类型的对应关系。为此，项自身必须包含类型；这一点通过一个内部类型片段图 ($\mathcal{T}$，未图示)实现，该图包含零个或多个类型的顶点和边的并集。特别是，其驻留关系 (R_R($\mathcal{T}$，未图示)可能是森林而非树。$\mathcal{I}$($\mathcal{T}$) 中作为该森林根的接口与 let 绑定构成双射 (R_{DI})。最后，内部对应关系 (R_{DC}，通过端口的相对定位表示)是一种对应关系，将每个 let 绑定的主体和各个出现与其关联的接口联系起来。现在，每个出现节点上的端口都与主体盒上的一个端口相关联，因为它们在内部类型中对应于同一个共享字段。

项之外的类型。虽然由 let 绑定派生的分量参与内部对应关系，但由项的根框派生的分量不参与。给定一个类型T 和一个项 t，C 是 t 与 T 之间的外部对应关系，当且仅当 C 是一个对应关系，C将 t的根与 T的根关联起来，并且 C与 R_{DC}不相交。当一个项与一个外部对应关系配对时，每个框、节点和端口（直接连接到 let 绑定的端口除外）都恰好对应一个接口或字段。

项的正则性。函数式语言中类型为 T 的项 t，翻译后不仅是一个 Ideograph 项，而且是一个 Ideograph 项 $(\!|t|\!)_G$ 与一个 $(\!|t|\!)_G$ 和 $(\!|T|\!)$之间的外部对应关系 $(\!|t|\!)_C$组成的对。请看图1B。在传统函数式语言中，这个类型有两个线性项：`id := fun x, y => (x, y)` 和 `swap := fun x, y => (y, x)`。在 Ideograph 中，只有一个项，如图所示，它等于 $(\!|\texttt{id}|\!)_G = (\!|\texttt{swap}|\!)_G$。然而，该项与类型 $(\!|\texttt{id}|\!)_C \neq (\!|\texttt{swap}|\!)_C$之间存在两个不同的外部对应关系。在递归翻译项时，内部对应关系会相互抵消，因此我们同时得到$(\!|\texttt{id a b}|\!)_G = (\!|\texttt{swap b a}|\!)_G$ 和 $(\!|\texttt{id a b}|\!)_C = (\!|\texttt{swap b a}|\!)_C$。这样便对内部参数顺序取商，同时允许在外部区分 `id` 和 `swap`。

2.4 连通关系与项的良构性

如前所述，类型具有连通关系 R_C。这是每个接口中各字段之间的一种对称、反自反关系。直观地说，该关系表示节点“内部”端口之间允许的连接，从而表示替换节点后哪些端口可能被连线。这一条件具有多项优势：它排除了可能导致无界递归的自指 let 绑定；它使我们能够忠实地表示传统函数，其中函数的输出不能作为输入反馈回来；此外，若没有这一条件，我们对数据结构的表示将无法正确工作（见图9 (iv) 和图 11 (iii)）。

粗略地说，连通关系区分了函数和积（更准确地说，在线性逻辑中 $\otimes$和 $\parr$ [11],，其中线性函数为 $A \multimap B := A^\perp \parr B$）。函数可以利用其输入生成输出（其输入端口和输出端口可以相连），而积必须分别生成其左侧和右侧（其左端口和右端口不能相连）。对于类型 T，定义其对偶 $T^\perp$：它具有相同的字段，但在顶层将接收端和提供端互换，并采用互补的连通关系。对于类型 T和 S，定义 $T \sqcup S$为包含 T 和 S 的字段的不交并，以及它们的连通关系的不交并。定义 $T \bowtie S$ 为 $T \sqcup S$，但在 T 的顶层字段与 S的顶层字段之间添加连通边。将普通函数式类型转换为 Ideograph 时，$(\!|A \times B|\!)$ 变为 $(\!|A|\!) \sqcup (\!|B|\!)$，而 $(\!|A \to B|\!)$ 变为 $(\!|A|\!)^\perp \bowtie (\!|B|\!)$。示例见图 5。

要使一个项格式良好，其布线关系与其类型所给出的连通关系的组合必须是无环的，但有某些例外（见定义 13）。图 6 展示了格式良好和格式不良的项片段。这里有四个片段，分别结合两种不同的类型来考察。顶行的类型类似于 `X -> X`。C 有效，类似于普通函数调用；E 无效，类似于将输出反馈为输入的调用；G 有效，类似于一个函数：它将参数传递给另一个函数以丢弃该参数，然后返回一次独立函数调用的结果；

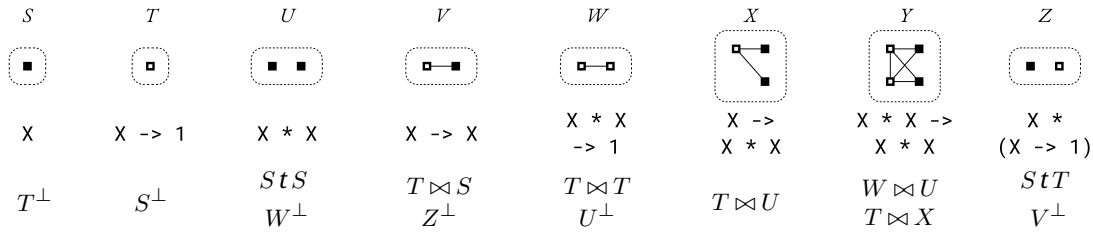


图 5：每列从上到下依次为：一个名称；一种类型；一个函数类比；以及一种或两种使用$\sqcup$、 $\bowtie$和$(\cdot)^\perp$从其他类型构造该类型的方式。

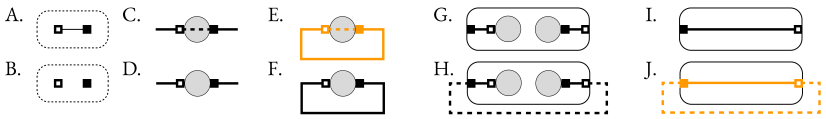
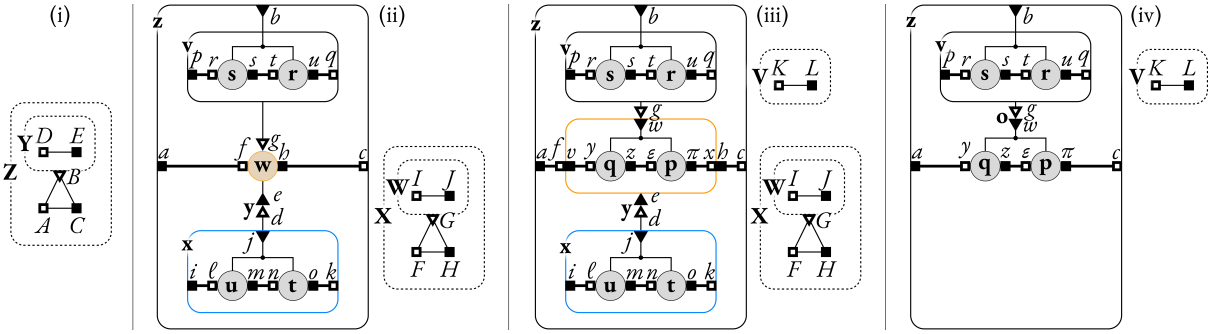


图 6：类型（A、B）和项片段（C、D、E、F、G、H、I、J）。A 中的接口对应于 C 和 E 中的节点，以及 G 和 I 中的框。B 中的接口对应于 D 和 F 中的节点，以及 H 和 J 中的框。虚线不属于项，但表示 R_C 类型的字段之间的关系，该关系显示在对应的端口之间。请注意，由于框使用对偶类型，G、H、I 和 J 中的虚线与 R_C互补。E 和 J 是格式错误的项，因为其中存在橙色高亮显示的环。

I 有效，类似于一个将输入作为输出返回的函数。底行的类型在函数式编程中没有完全对应的形式，但大致对应于 `(X -> 1) * X`：一个接受 `X` 的延续，以及一个类型为 `X`的值。D 有效，类似于分别使用延续和值；F 有效，类似于将值传递给延续；H 有效，类似于分别通过函数调用构造延续和值；J 无效，类似于将最终传递给延续的参数作为该对的右侧返回。E 和 J 都存在禁止的环。



(v)

$I(T) = \{\mathbf{X}, \mathbf{W}\}$
$F(T) = \{F, G, H, I, J\}$
$D = \{\mathbf{y}\}$
$R_A = \{(e, \mathbf{y}), (d, \mathbf{y}) \ldots\}$
$R_{DI} = \{(\mathbf{y}, \mathbf{X})\}$
$R_{WC} = \{(b, g)\}$
$R_{CU} = \{(\mathbf{s}\ b), (\mathbf{r}\ b) \ldots\}$
$R_{DC} = \{(\mathbf{w}, \mathbf{X}), (g, G), (\mathbf{v}, \mathbf{W}), (\mathbf{x}, \mathbf{X}), (j, G), (\mathbf{u}, \mathbf{W}), (\mathbf{t}, \mathbf{W}) \ldots\}$
$C = \{(\mathbf{z}, \mathbf{Z}), (b, B), (\mathbf{s}, \mathbf{Y}), (\mathbf{r}, \mathbf{Y}) \ldots\}$

(vi)

$I(T) = \{\mathbf{V}\}$
$F(T) = \{K, L\}$
$D = \{\mathbf{o}\}$
$R_A = \{(g, \mathbf{o}), (w, \mathbf{o}) \ldots\}$
$R_{DI} = \{(\mathbf{o}, \mathbf{V})\}$
$R_{WC} = \{(b, g)\}$
$R_{CU} = \{(\mathbf{的}\ b), (\mathbf{的}\ b) \ldots\}$
$R_{DC} = \{(\mathbf{v}, \mathbf{V}), (\mathbf{q}, \mathbf{V}), (\mathbf{p}, \mathbf{V}) \ldots\}$
$C = \{(\mathbf{z}, \mathbf{Z}), (b, B), (\mathbf{s}, \mathbf{Y}), (\mathbf{r}, \mathbf{Y}) \ldots\ldots\}$

图7：类型 (i) 下操作语义的一个带注释的步骤，从项 (ii) 到项 (iv)（此前已在图 3 中展示）。这是一个中间步骤的示例，它不是项 (iii)。前项 (v) 和后项 (vi) 的部分形式化，以及它们与 (i) 中类型的对应关系 C。形式化中省略的部分与图 4 中的部分类似。内部类型 $\mathcal{T}$通常不会被描绘，但此处将其显示出来。项 (ii) 包含一个内部接口 **X**，它是 let 绑定 **y** 的类型。在 **y** 处执行步进，会将 **x** 的内容（**y** 的主体）替换到 **w** （**y** 的出现）的位置，并将 **W** 复制到 **V** （如 (iii) 所示）。最后，资源端口对 f、v 和 x、h 被连线替代，而 g、w 和 **V** 被附加到一个新的 let 绑定 **o** 上，从而得到项 (iv)。

2.5 形式体系

现在，我们对 Ideograph 给出一个精确的表述。我们建议在引入定义时参考图4和图7，以便理解这些定义。

定义 1（片段图）。 我们在图 8 中定义类型片段图 和项片段图 。每个图都由若干顶点集合和若干带有条件的边关系组成。

定义 2（类型和项）。 如果 R_R 具有单个根接口，则类型片段图是类型 。如果 R_R 具有单个根框，则项片段图是项 。

分量	Kind	描述
$\mathcal{I}$	set	接口，以虚线圆角矩形表示
$\mathcal{F}$	set	字段，分为构造器（$\mathcal{F}_C$，三角形）或资源（$\mathcal{F}_R$，正方形）以及提供（$\mathcal{F}_+$，实心）或接收（$\mathcal{F}_-$，空心）
R_R	$(\mathcal{I} \rightharpoonup \mathcal{I}) \otimes (\mathcal{F} \to \mathcal{I})$，无环的	归属关系，以包含在接口中的方式表示
R_I	$\bigotimes_{i \in \mathcal{I}} F^i_C \leftrightarrow I^i$	构造器-接口关联，通过字段与接口之间的接触描绘字段与接口之间的接触
R_C	$\bigotimes_{i \in \mathcal{I}}$ 余图$\mathcal{F}^i$	连通性，以实线表示；余图 是定义3
$\mathcal{B}$	set	框，以实心圆角矩形表示
$\mathcal{N}$	set	节点，表示为灰色圆形或矩形
$\mathcal{P}$	set	端口，分为构造器（$\mathcal{P}_C$，三角形）或资源（$\mathcal{P}_R$，正方形），以及提供（$\mathcal{P}_+$，实心）或接收（$\mathcal{P}_-$，空心）
$\mathcal{D}$	set	*let* 绑定，表示为构造器端口之间的接触
$\mathcal{T}$	类型片段图	内部类型片段图，不予表示
R_R	$(B \rightharpoonup B) \otimes (N \cup P \cup D \to B)$，无环的	归属关系，用框中的包含关系表示
R_A	$\bigotimes_{b \in \mathcal{B}} P^b \to (N^b \cup D^b \cup \{b\})$	附着关系，表示为之间的接触端口和节点/盒
R_{WR}	$\bigotimes_{b \in \mathcal{B}} P^b_{R+} \leftrightarrow P^b_{R-}$	资源连接，用粗线表示
R_{WC}	$\bigotimes_{b \in \mathcal{B}} \mathcal{P}^b_{C+} \asymp^P {}^b_{C-}$	构造器连接，未表示
R_{CA}	$\bigotimes_{b \in \mathcal{B}} B^b \leftrightarrow P^b_{C-}$	构造器参数，用细线表示
R_{CU}	$\bigotimes_{b \in \mathcal{B}} N^b \to \bigcup_{b \sqsubseteq b'} \mathcal{P}^{b'}_{C+}$，连线安全	构造器使用，用细线表示，可能带有分支节点连线；连线安全 是定义 4
R_{DI}	$\mathcal{D} \leftrightarrow \mathcal{I}(\mathcal{T})$的根	*let* 绑定类型推导，未描绘
R_{DC}	$\bigotimes_{d \in \mathcal{D}}$ d的对应关系	*let* 绑定对应关系，通过以下内容描绘盒和节点上的端口；的对应关系是定义 6

图 8：类型片段图（上）和项片段图（下）的分量。R $\leftrightarrow$S 是一对一关系，R $\to$S 是多对一关系，R $\rightharpoonup$ S 是多对一或零关系，R $\asymp$S 是多对多关系。S $\otimes$R $=\{S \cup R\colon S \in$S $R \in$R$\}$。S^{i}和 S^{b}表示 S 中分别直接归属于 i 和 b 的子集。 cv i和cv b表示 c 在驻留森林中位于 i 和 b 之下。F$_{C-}$ = F$_C \cap$F$_-$，F$_{C+}$、F$_{R-}$、F$_{R+}$、P$_{C-}$、P$_{C+}$、P$_{R-}$和 P$_{R+}$ 也同理。

定义3（余图）。由顶点为 $\mathcal{V}$ 的对称无自反图构成的集合 V 上的余图 是包含单点图且在补图和不交并运算下封闭的最小集合。直观而言，这是由原子$\mathcal{V}$ 通过合取和析取构成、并对结合律和交换律取商后得到的公式集合。

定义 4（连线安全性）。 假设存在构造器使用关系 R_{CU}、构造器连线关系 R_{WC} 以及一对 $(n,\ c) \in R_{CU}$。令 c位于 b_c中，且 n位于 b_n中，其中 $b_n \sqsubseteq b_c$。如果$b_n \neq b_c$，令 b'_n为直接位于 b_c中的盒，使得 $b_n \sqsubseteq b'_n \sqsubset b_c$，并令 c'为与 b'_n关联的构造器接收端口（位于 b_c中）。对于 R_{WC}， R_{CU}是连线安全的，当且仅当对于所有 $(n,\ c) \in R_{CU}$，要么$b_n = b_c$，要么 $(c,\ c') \in R_{WC}$。

定义 5（对应关系）。 给定一个类型 T和一个项 t，若关系 $C \in (\mathcal{B} \rightharpoonup \mathcal{I}) \otimes (\mathcal{N} \rightharpoonup \mathcal{I}) \otimes (\mathcal{P} \rightharpoonup \mathcal{F})$ 满足以下条件，则称其为一个对应关系：(1) 如果 $b \in \mathcal{B}$（或 $n \in \mathcal{N}$）与$\iota \in \mathcal{I}$对应，则 b（或 n）所连接的端口与 ι的字段之间的对应关系是双射。(2) 如果 $p \in \mathcal{P}_C$ 与 $f \in \mathcal{F}_C$对应，则与 p 关联的盒与和 f关联的接口对应。(3) 构造器端口和资源端口分别与构造器字段和资源字段关联。(4) 如果 p 与 f对应，则当 p 连接到节点时，它们的接收端和提供端种类相同；当 p 连接到盒时，它们的接收端和提供端种类相反。

定义 6（let 绑定对应关系）。 若对应关系 C 是 $d \in \mathcal{D}$的对应关系，则盒 d（通过 R_A 和 R_{CA}）的盒以及 d（通过 R_A 和 R_{CU}）的节点，分别对应于 d（通过 R_{DI}）的接口。不同的 let 绑定对应关系必须覆盖不相交的项组件集合。

定义 7（外部对应关系）。 给定类型 T 和项 t，如果对应关系 C 是 t 与 T 之间的外部对应关系，则称之为外部对应关系，其中 C 将 t的根框与 T 的根接口关联起来，并且 C 与 R_{DC}不相交。

备注。 给定类型 T、项 t以及 t 与 T之间的外部对应关系 C。令$C^* := C \cup R_{DC}$。每个 $n \in \mathcal{N}$和 $b \in \mathcal{B}$在 C^*中恰好出现一次。对于每个 $p \in \mathcal{P}$，它要么附加到某个 $d \in \mathcal{D}$ 上且不出现在 C^*中，要么在 C^*中恰好出现一次。

定义 8（项相等）。 给定一个类型 T、项 t_1 和 t_2，以及 t_1 与 T 之间的对应关系 C_1 和 t_2 与 T 之间的对应关系 C_2，如果固定顶点的具体标记以得到 $\widehat{T}$、 $\widehat{t_1}$、 $\widehat{t_2}$、 $\widehat{C_1}$ 和 $\widehat{C_2}$，存在对 $\widehat{t_2}$ 中顶点的某个重新标记 h，使得 $(\widehat{t_1},\ \widehat{C_1}) = (h\ (\widehat{t_2})\ \ h\ (\widehat{C_2}))$，则称 $(t_1,\ C_1)$ T-相等 于 $(t_2,\ C_2)$。

定义 9（替换）。假设有类型 T、项 t，以及 C 在 t 与 T之间的对应关系。给定 $b \in$ B 和 $n \in$ N，其中 b 归属于某个 $b_0 \in$ B，且 n 位于驻留森林中 b_0 所在位置或其下方；再给定 $\iota \in$ I(T)，以及 $C_b \subseteq R_{DC}$ 在 b 与 ι 之间、 $C_n \subseteq R_{DC}$在 n 与 ι之间的对应关系，我们将用(b, C_b)在 ι 处替换(n, C_n)定义为执行以下操作的结果：(1) 删除 n （从 N 及所有关系中删除）。(2) 对每个归属于 b的分量，创建一个新副本，使其归属于原先包含 n的盒，同时在 R_{DC} 和 C中也创建相应副本。(3) 对每个连接到 b的端口 p_b ，令 p'_b 为 p_b的新副本，并令 $C'_{bp} \subseteq R_{DC}$ 为与 p'_b相关的部分。注意，对于每个归属于 ι的 f ，现在都有一个新鲜的 p'_b ，以及一个原先连接到 n的 p_n ，两者构成一对接收/提供对。令 $C_{np} \subseteq C_n$ 为与 p_n相关的部分。(4) 对每个资源字段 f， p'_b 通过连线连接到某个 p''_b ，而 p_n 通过连线连接到某个p'_n。擦除 f、 p'_b和 p_n，并将(p''_b, p'_n)添加到布线关系中。(5) 对每个构造器字段f，创建一个新的 let 绑定 d，并将 p'_b和 p_n连接到它。对于与 f 关联的 $\iota_f \in$ I(T)，在 I(T) 中为其子树创建一个新副本，并令该副本的根为 ι'_f。将 C'_{bp}和 C_{np}改为指向 ι'_f。将(d, ι'_f)添加到 R_{DI}。

定义 10（let 绑定内联）。假设有类型 T、项 t，以及 C在 t 与 T之间的对应关系。给定一个 let 绑定 $d \in \mathcal{D}$，令其关联接口为 ι，连接到它的构造器接收端口为 p_-，连接到它的构造器提供端口为 p_+，传递给p_- 的参数为盒 b，由 p_+ 生成的节点集合为 N，R_{DC} 中与 b 相关的子集为 C_b，并且对每个 $n \in N$，R_{DC} 中与 n 相关的子集为 C_n。将 d 的内联定义为执行以下操作的结果：(1) 对每个 $n \in N$，在 ι处用(b, C_b)替换(n, C_n)。(2) 删除 d、ι、p_-、p_+和 b。

定义 11（归约）。给定类型 T、项 t_1 和 t_2，以及 C_1 在t_1与 T之间、C_2在 t_2与 T之间的对应关系，如果存在一个$d \in \mathcal{D}(t_1)$，使得将 d内联到 (t_1, C_1) 中的结果与 (t_2, C_2) T-相等，则称 (t_1, C_1) T-归约 为 (t_2, C_2)。

定义 12（组件下降关系）。组件 是盒、节点、端口或 let 绑定。若组件 c_1 是 c_2 的子级，则 c_1 附着于（在 R_A 中相关于）c_2；或者 c_1 是 c_2的构造器参数（在 R_{CA} 中相关于）；或者 c_1 是 c_2的构造器使用（在 R_{CU} 中相关于）。对于这一关系的传递闭包，c_1 从 c_2下降。请注意，下降关系形成一个独立于驻留森林（R_R）的森林；还请注意，每个组件要么从一个 let 绑定下降，要么从驻留森林的某个根节点下降。

定义 13（良构性）。设有一个类型 T、一个项 t，以及 t 与 T 之间的外部对应关系C。令 W 为端口 $R_{WR} \cup R_{WC}$ 上的关系。定义端口上的关系 F，使得（p_1，p_2）$\in F$，当且仅当满足以下任一条件：（1）p_1 和 p_2 附加到同一个 $d \in$ D；（2）令 c 为 p_1 和 p_2 在下降关系中的最近公共祖先盒或节点，令 p_1'、p_2' 分别为附加到 c 的 p_1、p_2 的祖先，并令 f_1'、f_2' 分别为与 p_1'、p_2' 对应的字段；若不存在这样的 c，则（p_1，p_2）$\notin F$；否则，若 c 是节点，则当且仅当（f_1'，f_2'）$\in R_C$ 时，（p_1，p_2）$\in F$；若 c 是盒，则当且仅当（f_1'，f_2'）$\notin R_C$ 时，（p_1，p_2）$\in F$。现在，如果对于 F 和 W 中交替取边的每个环，该环中都存在一对端口 p_1、p_2，使得（p_1，p_2）$\in F$，但边（p_1，p_2）不属于该环，则称（T，t，C）是良构的。（这直接受到 [25]，中带弦无环 R&B-余图条件的启发，并扩展了该条件以处理接口嵌套。）

3 表达数据

3.1 二叉树

现在我们将以无标号二叉树为例，了解 Ideograph 如何表示数据。在多态λ演算中，它们表示为类型 $\forall X.(X \to X \to X) \to (1 \to X) \to X$：第一个参数 $X \to X \to X$ 是（二元）分支构造器，第二个

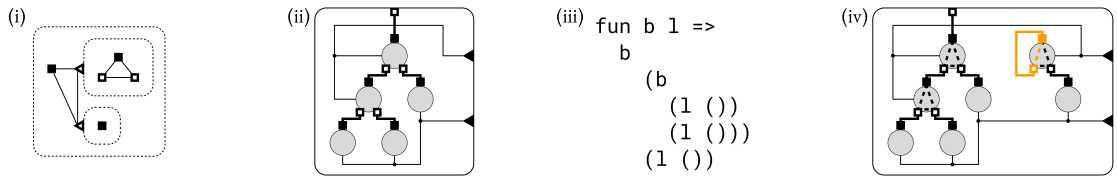


图9：无标号二叉树的类型（i）。该类型的一个项（ii）。该项在通用函数式语言中的丘奇编码表示（iii）。一个格式错误、不能作为无标号二叉树的项（iv）。虚线并非项的一部分，而是 R_C 位于类型字段之间，并显示在对应的端口之间。非法循环以橙色标出。

参数 $1 \to X$ 是（一元）叶构造器（由于树无标号，因此接收 unit）。去除多态性后，这对应于图9 (i) 中的 Ideograph 类型。右上方的构造器字段表示分支节点，其中包含用于一个父级（顶部）、一个左子节点（左下方）和一个右子节点（右下方）的资源端口；右下方的构造器字段表示叶节点，其中包含用于一个父级（顶部）的资源端口。剩余的资源端口（左上方）对应于树的根节点。

图 9 (ii) 和 (iii) 表示同一棵二叉树，其中有两个分支节点和三个叶节点。连通关系和良构条件排除了图 9 (iv) 中这类项。线性确保只有一棵树：额外的树没有资源端口可作为其根节点，因此这类项会被资源布线关系的双射性排除。

3.2 有向多重图

作为一种非项代数结构（因此缺少传统的 Church 编码） [5], ，我们来看有向多重图。它们在 Ideograph 中由图10 (i) 中的类型指定，其中右上方的字段对应“顶点”，右下方的字段对应“边”。图10 (ii) 展示了一个包含三个顶点和三条边的多重图，以及它作为一个项的表示。由于顶点可以关联任意数量的边，顶点节点使用构造器端口，而非资源端口。边有两个端口，用于接收来自其源顶点和目标顶点的构造器。这里，构造器提供端口直接连接到构造器接收端口；这种做法在形式上并不允许——为了清晰起见，我们滥用了记法，实际意指接收端口连接到一个盒，该盒包含一个由提供端口构造的单一节点。图10 (iii) 展示了一个类似的多重图，它有三个顶点，但包含六条边。还有一种用于表示多重集的相关类型，本质上就是没有边的多重图。

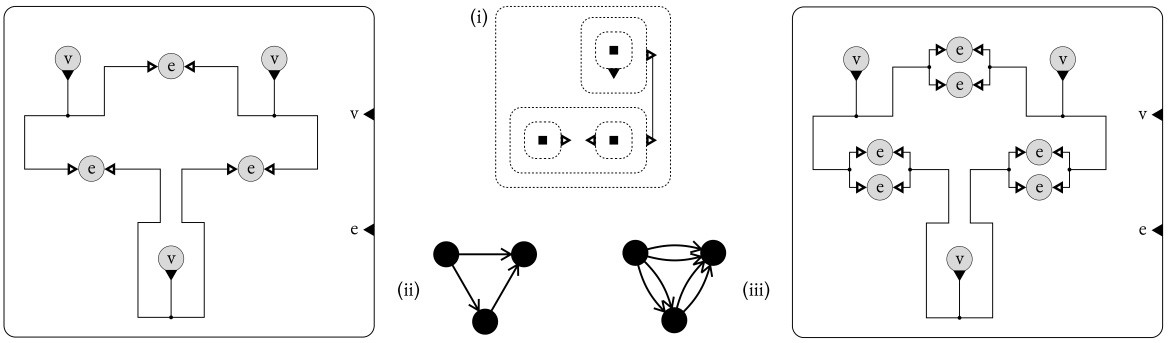


图10：有向多重图的类型（i）。两个项及其表示的有向多重图（ii）和（iii）。为提高可读性，构造器使用关系用“v”和“e”标签表示，而不是用线表示。

3.3 无类型λ演算

无类型 λ 演算中的闭项也不构成项代数。它们有三种节点——应用、抽象和变量——但每个变量节点都必须以某种方式与其上方的一个抽象相关联。

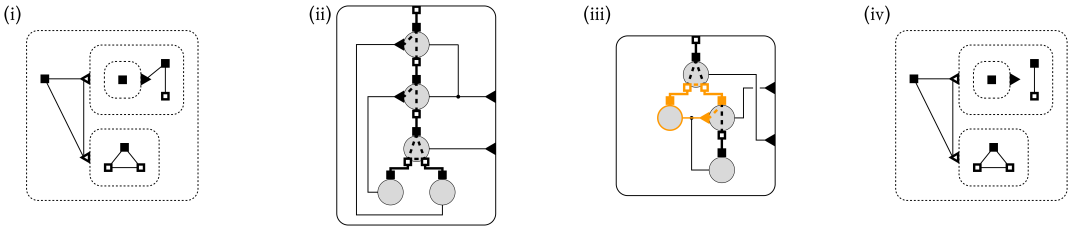


图 11：无类型 λ 演算中闭项的类型（i）。表示“$\lambda x.\lambda y.\, y\ x$”的项（ii）。对应于“$x\ (\lambda x.x)$”的项（iii）。另一种类型，与（i）仅相差一条连通边（iv）。将（i）中的连通关系以虚线叠加在（ii）和（iii）中对应的端口上。以橙色标记的环使（i）与（iii）之间的外部对应关系不合乎规范。（iii）与（iv）之间存在合乎规范的外部对应关系。

图 11 (i) 展示了它们在 Ideograph 中的类型，其中右下方的构造器字段表示应用，包含表示父节点（顶部）、左子节点（左下方）和右子节点（右下方）的资源字段；右上方的构造器字段表示抽象，包含表示父节点（顶部）和子节点（底部）的资源字段，以及一个表示“引用此抽象的变量节点”的构造器端口（左侧）。我们没有从一个用于“变量”节点的构造器开始，而是在每次构造抽象节点时，都会出现一个新的“变量”构造器。抽象接口上的连通关系会阻止变量出现在其绑定符上方，如图 11 (iii) 所示：“变量”构造器端口与“父节点”端口之间的边禁止了这种情况，而“变量”构造器端口与“子节点”端口之间没有边，因此变量可以出现在抽象的主体中。相比之下，图 11 (iv) 缺少变量—父节点边，因此允许出现这个项。

这种表示法与参数化高阶抽象语法（phoas） [28,6], 密切相关，后者利用参数多态来表示变量绑定。对我们的表示法进行的计算与 phoas 一样遵循项的绑定结构，从而能够实现单步 β 归约，并免费提供避免捕获的替换。

4 操作数据

现在我们展示如何操作这类数据结构。在这个简化的介绍中，由于缺乏多态性，我们只能举一个简单的示例，因为要在不使用复杂类型实例化全称量词的情况下，编写许多操作丘奇编码的函数，并不容易理解。尽管完整版本的 Ideograph 能够表示丰富得多的函数，下面的示例应该

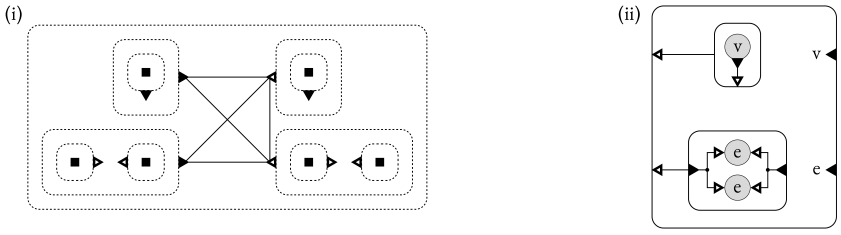


图12：从有向多重图到有向多重图的函数类型（i）。该类型的项会将输入中的每条边替换为一对边（ii）。

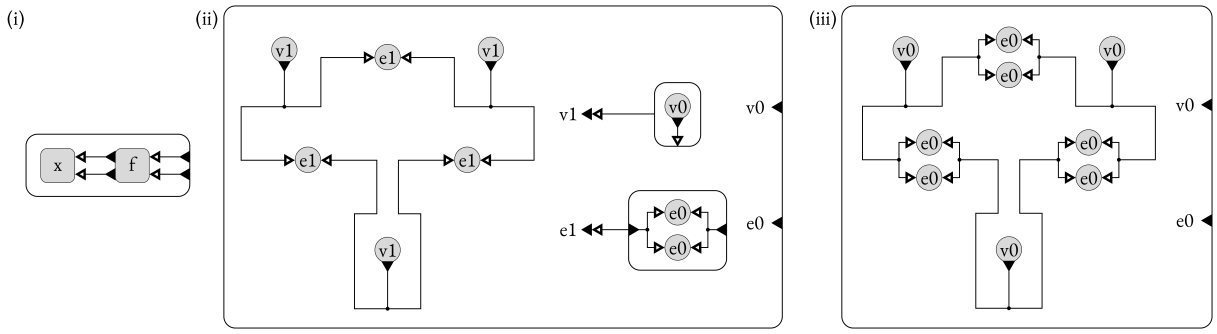


图 13：将“x”的实例传递给对“f”的调用的项 (i)。(i) 中内联“x”和“f”的定义后的结果 (ii)。(ii) 中内联 let 绑定“v1”和“e1”后的结果 (iii)。“x”是一个 let 绑定，其主体是图10 (ii) 中的有向多重图；“f”是一个 let 绑定，其主体是图12 (ii) 中的函数。(iii) 是图10 (iii) 中的项。

能提供正确的直觉。

回顾第3.2节中有向多重图的表示。图12 (ii) 中的项表现得像一个函数，它以有向多重图作为输入，并将每条边加倍。图13展示了该函数对图10 (ii) 中包含三个顶点和三条边的有向多重图进行求值的过程。

我们强调，在大多数编程语言中，操作图需要操作带标签节点的结构——无论是邻接矩阵，还是节点索引对列表——这使得我们可以编写依赖于标记的函数，因此这些函数并非真正作用于图的函数。Ideograph 无法做到这一点：它只是用某种模式替换结构中的每个节点（在本例中，顶点和边都是节点），就像丘奇编码一样；在这种情况下，每个顶点都被替换为一个顶点，每条边都被替换为两条方向相同的边。

遗憾的是，这种做法是保守的：有些合理的图上函数无法表达，其中包括返回顶点数量的丘奇数的函数。具体而言，Ideograph 有两种不同的类型与自然数相似，大致分别是“单位列表”和“单位多重集”；我们可以编写一个将顶点数量计为单位多重集的函数，却无法编写一个将单位多重集转换为单位列表的函数，尽管这种转换是健全的。虽然我们可以添加一个原语函数来实现这一点，但也可以考虑在内部定义它。如何刻画并扩展可表示函数的集合，将留待未来工作。

5 相关工作

线性逻辑。 Ideograph 与线性逻辑 [11]密切相关。线性逻辑的大多数论述都使用“收缩”“弱化”和指数模态的“去指数化”规则，但 Andreoli 的等价二元系统 [3] 改为使用一种称为“吸附”的规则，这与我们的节点和构造器使用关系非常相似。一个显而易见的区别是，我们的命题是图而不是树，因此可以将某些类型等价（例如积的排序）商掉。我们所商掉的类型等价类似于直觉主义类型系统 [8]的可证明的类型同构 。这使我们推测 Ideograph 是

带有混合规则的多态（二阶命题）乘法指数线性逻辑，但将这些类型同构商掉了。

证明网与交互网。我们的工作与证明网 [11], 密切相关，尤其是与其扩展形式——交互网 [15]。我们的工作与交互网之间有两个关键区别：（1）在交互网中，每个符号（大致相当于我们的“节点”）都有一个主端口，用于归约；而在我们的工作中，节点没有特权端口，归约完全通过将定义（类型或项的定义）代入其出现处来进行。（2）在交互网中，符号集合及其关联的端口必须预先固定；而在我们的工作中，符号集合并非固定，符号的出现可能会向集合中添加新符号。

已有研究使用交互网表示λ演算项 [19, 18, 21]。这项工作使用显式的“复制”和“擦除”符号，而指数模态（按我们的术语称为“构造器”）则是我们形式体系的核心部分。该工作的一个关键优势是，通过共享子项 [18]，在某些基准测试上提高了归约性能。我们希望在未来工作中使用他们的基准测试评估我们的系统。

证明网 [11] 和交互网 [16, 17] 都有一些版本用“盒”来表示指数模态，我们预计这些版本与 Ideograph 密切相关，尽管我们的类型是图而非树。

函数式编程。Ideograph 与 OCaml、Haskell 和 Rust 等较传统的函数式编程语言有几项关键区别。(1) Ideograph 没有原始归纳数据类型，而是使用丘奇编码的类似形式。(2) I deograph 既是纯粹的，也是强正规化的，并且不规定求值顺序。(3) 按照 Girard 的定义，Ideograph 是线性的 [11], ；而 Rust 和线性 Haskell 缺少指数模态，线性 Haskell 有单独的非线性类型，Rust 则是仿射的。(4) 大多数语言中的函数会隐式返回单个值，而 Ideograph 中的盒（“函数”）会显式命名其零个或多个输出，这与 C 及类似语言中的输出参数相似。(5) Ideograph 可以自然地解释为图替换，即便人们更倾向于用文本形式体系编写程序。

多态λ演算。（多态）Ideograph 与多态λ演算（System F）之间有三个关键区别：据推测，Ideograph 是带有混合规则的多态乘法指数线性逻辑（pmell）的规范版本；pmell 是多态直觉主义线性逻辑（pill）的经典对应形式；而 pill 是 System F 的线性对应形式。此处的表述不是多态的，因此对应于直觉主义线性逻辑和简单类型 λ 演算。

在表达数据类型的能力方面，我们预期 Ideograph 和 pmell 是相同的。然而，Ideograph 的正则性意味着，例如，对于有向多重图 g，在 pmell 中， g的顶点和边的每一种标记方式都对应一个不同的（完全规范化的）项，而在 Ideograph 中，则只有一个（完全规范化的）项来表示 g。我们不确定线性和经典性会如何影响表达数据的能力（即类型集合及其完全规范化项）。线性和经典性对语言的计算行为有公认的影响：线性演算通常能够明确区分值调用和名调用 [14], ，而经典性则允许表达 call/cc [13] 之类的构造。构成异构项代数的数据结构可以通过过程化的丘奇编码转换为

System F 类型 [5], ，但有向多重图和 λ 演算项之类的结构并不是项代数。

编程语言的图表示。已有研究将现有编程语言，尤其是 λ 演算，表示为图 [10, 26, 12]。我们的工作与之一个关键区别在于，我们并非试图表示现有编程语言，以用于优化编译等目的。相反，我们希望表示数据结构（语法树恰好是其中一种）以及对这些数据结构进行的纯计算。因此，我们不关注副作用或求值顺序，而这些正是许多相关研究所着力处理的问题。

类型的图表示。已有研究将乘法线性逻辑中的公式表示为无向图 [2, 25]。当我们不使用指数模态或二阶命题量词时，我们的类型系统与这些研究密切相关。（注意，对于 $\otimes$ 和 $\parr$，我们采用了与他们相反的边约定。）

图的表示。已有研究通过递归绑定器，在纯函数式编程语言中表示图结构 [20]。虽然我们的系统采用线性类型，而他们的系统并非如此，但我们认为两者关系密切，并希望在未来工作中深入探讨这种联系。

图编程语言。 有多种图编程语言，包括 GROOVE [24], GP-2 [23], 和 LMNtal [27]。我们所知的这类系统都包含重写规则的概念：该规则匹配一个子图，并将其替换为另一个子图。相比之下，我们的系统只有一条归约规则，类似于 β归约。LMNtal 没有类型系统，而类型是 Ideograph 的核心组成部分。HyperLMNtal [29], 通过为 LMNtal 扩展超边，曾被用于编码 λ演算项。与我们使用构造器的方式不同，他们通过一条超边将绑定符连接到其所有变量出现。Ideograph 是“带标记端口图” [9], 的一种变体。这种形式体系中，边通过“端口”连接到节点，且已被用于表示程序。

参数化高阶抽象语法。 有研究使用元语言中的函数来编码带变量绑定的语法。具体而言， [28] 和 [6] 利用参数多态性，精确编码多种 λ演算中的闭项，并且只允许进行保持结构的操作。将它们的编码转换为 Ideograph 的类型后，与我们在第 3.3 节中提出的类型非常接近。不过，Ideograph 可以表示变量必须恰好使用一次的语言，而参数化高阶抽象语法则不能。此外，我们的目标是表示语法以外的结构。

6 未来工作

未来工作有几个方向。我们首先必须证明 Ideograph 的标准性质，包括主谓约简和强正规化。接下来，我们希望证明项等价是图同构完备的，刻画类型所能表示的结构，并正式确立它与线性逻辑之间的联系。最后，我们计划开发一个实现；由于操作语义简单，我们预计这项工作会相当直接。

致谢

我们感谢 Ian Mackie、Kazunori Ueda 和两位匿名审稿人提供的宝贵反馈，也感谢 Lawrence Dunn、Harrison Goldstein、Eleftherios Ioannidis、Nick Rioux 和 Lucas Silver 阅读早期稿件。本研究部分由美国国家科学基金会（NSF）资助，资助编号为 CCF-1910769 和 CCF-1917852。

参考文献

[1] Samson Abramsky（1993）：线性逻辑的计算解释。理论计算机科学 111(1)，第 3–57 页，doi:10.1016/0304-3975(93)90181-R。[2] Matteo Acclavio、Ross Horne 与 Lutz Straßburger（2020）：超越公式的逻辑：基于图的证明系统。载于：第 35 届美国计算机协会/电气电子工程师学会逻辑与计算机科学研讨会论文集，LICS '20，美国计算机协会，美国纽约州纽约，p. 38–52，doi:10.1145/3373718.3394763。[3] Jean-Marc Andreoli（1992）：线性逻辑中的聚焦证明与逻辑编程。逻辑与计算杂志 2(3)，第 297–347 页，doi:10.1093/logcom/2.3.297。[4] Brian E. Aydemir、Aaron Bohannon、Matthew Fairbairn、J. Nathan Foster、Benjamin C. Pierce、Peter Sewell、Dimitrios Vytiniotis、Geoffrey Washburn、Stephanie Weirich 与 Steve Zdancewic（2005）：面向大众的元理论机械化：PoplMark 挑战。载于：第 18 届高阶逻辑定理证明国际会议论文集，TPHOLs'05，施普林格出版社，柏林、海德堡，p. 50–65，doi:10.1007/11541868_4。[5] Corrado Böhm 与 Alessandro Berarducci（1985）：项代数上类型化 Λ-程序的自动综合。理论计算机科学 39，第 135–154 页，doi:10.1016/0304-3975(85)90135-5。软件技术与理论计算机科学基础第三届会议。[6]Adam Chlipala（2008）：用于语义机械化的参数化高阶抽象语法。载于：第 13 届美国计算机协会 SIGPLAN 函数式编程国际会议论文集，ICFP '08，美国计算机协会，美国纽约州纽约，p. 143–156，doi:10.1145/1411204.1411226。[7] N.G de Bruijn（1972）：使用无名占位符的 λ 演算记法：一种用于自动处理公式的工具，并应用于丘奇-罗瑟定理。数学研究（会议录）75(5)，第 381–392 页，doi:10.1016/1385-7258(72)90034-0。[8] R. Dicosmo（1995）：二阶同构类型：关于带满射配对和终对象的二阶λ-演算的证明论研究。信息与计算 119(2)，第 176–201 页，doi:10.1006/inco.1995.1085。[9] Maribel Fernández、Hélène Kirchner 与 Bruno Pinaud（2018）：带标签端口图——模型与计算的形式结构。理论计算机科学电子札记 338，第 3–21 页，doi:10.1016/j.entcs.2018.10.002。第 12 届逻辑与语义框架及其应用研讨会（LSFA 2017）。[10] Dan R. Ghica、Koko Muroya 与 Todd Waugh Ambridge（2019）：一种稳健的基于图的方法处理观察等价，doi:10.48550/ARXIV.1907.01257。[11]Jean-Yves Girard（1987）：线性逻辑。理论计算机科学 50(1)，第 1–101 页，doi:10.1016/0304-3975(87)90045-4。[12] Clemens Grabmayer（2018）：使用带结构约束的图对项建模（两个例证）。载于 Maribel Fernández 与 Ian Mackie 编：第十届项与图计算国际研讨会论文集，TERMGRAPH@FSCD 2018，英国牛津，2018 年 7 月 7 日，EPTCS 288，第 1–13 页，doi:10.4204/EPTCS.288.1。

[13] Timothy G. Griffin（1989）：公式即类型的控制观念。载于：第 17 届美国计算机协会 SIGPLAN-SIGACT 程序设计语言原理研讨会论文集，POPL '90，计算机协会，美国纽约州纽约，p. 47–58，doi:10.1145/96709.96714。[14] Giulio Guerrieri 与 Giulio Manzonetto（2018）：Bang 演算与 Girard 的两种翻译。载于 Thomas Ehrhard 、 Maribel Fernández 、 Valeria de Paiva 与 Lorenzo Tortora de Falco 编：线性与线性逻辑及其应用的发展趋势联合国际研讨会论文集，Linearity-TLLA@FLoC 2018，英国牛津，2018 年 7 月 7–8 日，EPTCS 292，第 15–30 页，doi:10.4204/EPTCS.292.2。[15] Yves Lafont（1989）：交互网。载于：第 17 届美国计算机协会 SIGPLAN-SIGACT 程序设计语言原理研讨会论文集，POPL '90，计算机协会，美国纽约州纽约，p. 95–108，doi:10.1145/96709.96718。[16] Yves Lafont（1995）：从证明网到交互网，p. 225–248。伦敦数学学会讲义系列，剑桥大学出版社，doi:10.1017/CBO9780511629150.012。[17] Ian Mackie（2000）：线性逻辑的交互网。理论计算机科学 247(1)，第 83–140 页，doi:10.1016/S0304-3975(00)00198-5。[18] Ian Mackie（2011）：闭归约的交互网实现。载于 Sven-Bodo Scholz 与 Olaf Chitil 编：函数式语言的实现与应用，施普林格出版社柏林海德堡，柏林、海德堡，第 43–59 页，doi:10.1007/978-3-642-24452-0_3。[19] Ian Craig Mackie（1994）：实现的几何学。博士论文，帝国理工学院，doi:10.25560/46072。[20]Bruno C.d.S. Oliveira 与 William R. Cook（2012）：使用结构化图进行函数式编程。载于：第 17 届美国计算机协会 SIGPLAN 函数式编程国际会议论文集，ICFP '12，美国计算机协会，美国纽约州纽约，p. 77–88，doi:10.1145/2364527.2364541。[21] Vincent van Oostrom、Kees Jan van de Looij 与 Marijn Zwitserlood（2004）：Lambdascope：λ演算的另一种最优实现。[22] Benjamin C. Pierce（2002）：类型与程序设计语言，第 1 版。麻省理工学院出版社。[23] Detlef Plump（2011）：GP 2的设计。载于 Santiago Escobar 编：第 10 届重写与编程中的归约策略国际研讨会论文集，WRS 2011，塞尔维亚诺维萨德，2011 年 5 月 29 日，EPTCS 82，第 1–16 页，doi:10.4204/EPTCS.82.1。[24] Arend Rensink（2004）：GROOVE模拟器：状态空间生成工具。载于 John L. Pfaltz、Manfred Nagl 与 Boris Böhlen 编：具有工业相关性的图变换应用，施普林格出版社柏林海德堡，柏林、海德堡，第 479–485 页，doi:10.1007/978-3-540-25959-6 40。_[25] Christian Retoré（2003）：精美的证明网：完美匹配与余图。理论计算机科学 294(3)，第 473–488 页，doi:10.1016/S0304-3975(01)00175-X。线性逻辑。[26] Ralf Schweimeier 与 Alan Jeffrey（1999）：闭包转换的范畴论与图论处理。理论计算机科学电子札记 20，第 481–511 页，doi:10.1016/S1571-0661(04)80090-2。MFPS XV，程序设计语义的数学基础，第十五届会议。[27] Kazunori Ueda（2009）：LMNtal：一种分层逻辑程序设计语言。理论计算机科学 410(46)，第 4784–4800 页，doi:10.1016/j.tcs.2009.07.043。抽象解释与逻辑编程：纪念 Giorgio Levi 教授。[28] Geoffrey Washburn 与 Stephanie Weirich（2003）：盒子疯狂了：使用参数多态编码高阶抽象语法。载于：第八届美国计算机协会 SIGPLAN 函数式编程国际会议论文集，ICFP '03，美国计算机协会，美国纽约州纽约，p. 249–262，doi:10.1145/944705.944728。

[29]Alimujiang Yasen 和 Kazunori Ueda（2021）：重新审视 *HyperLMNtal* 中的图类型：用于超图重写的建模语言。*IEEE Access* 9，第 133449–133460 页，doi:10.1109/ACCESS.2021.3112903。