Ideograph: A Language for Expressing and Manipulating Structured Data

Stephen Mell
University of Pennsylvania
sm1@cis.upenn.edu

Osbert Bastani
University of Pennsylvania
obastani@seas.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@seas.upenn.edu

We introduce Ideograph, a language for expressing and manipulating structured data. Its types describe kinds of structures, such as natural numbers, lists, multisets, binary trees, syntax trees with variable binding, directed multigraphs, and relational databases. Fully normalized terms of a type correspond exactly to members of the structure, analogous to a Church-encoding. Moreover, definable operations over these structures are guaranteed to respect the structures' equivalences. In this paper, we give the syntax and semantics of the non-polymorphic subset of Ideograph, and we demonstrate how it can represent and manipulate several interesting structures.

1 Introduction

Structured data is ubiquitous: lists, trees, graphs, relational databases, and syntax trees are just a few of the structures that underpin computer science. We often want to perform operations on such objects in ways that both respect and leverage their structure. For instance, we might wish to aggregate the elements of a bag (multiset). We could represent bags as lists and fold over them as lists, but this provides no guarantee that the result is invariant to the order. Or, we might wish to manipulate syntax trees of programs. We could represent variables as names or de Brujin indices [7], but in either case operations on the representation must be shown to respect the binding structure. Other, similar, circumstance arise often in practice.

Yet, there are surprisingly few formalisms for actually defining such structures, much less for defining invariant-respecting operations over them. Relational database schemas define bags of records, with certain additional structure (most notably, foreign key constraints between tables). Most widely used programming languages, like C, Java, and Python, and data interchange formats, like Google's Protocol Buffers, have limited type systems, supporting at most product, sum, and function types, but not supporting the graph structures and constraints that would be required to define bags or syntax trees with variable binding. Dependently-typed languages, like Coq, are capable of imposing complex constraints, but even simple data structures, like syntax trees with binding structure, have proven tricky to deal with in practice [4].

As a result, we resort to implementing ad-hoc solutions. For aggregating over bags, we can separately prove that our function is invariant to order, and thus is truly a function over bags rather than lists. For manipulating syntax trees, we can separately prove that our substitution operation is capture-avoiding. However, this must be done for each new structure and operation. We want a general formalism for representing and manipulating a rich class of structures.

Graphs are a common formalism for encoding many kinds of data, but they don't capture everything. For example, binary trees are "graphs", but they have more structure: they have two distinct kinds of nodes ("branch" and "leaf"), two kinds of edges ("left child" and "right

C. Grabmayer (Ed.): Post-Proceedings of TERMGRAPH 2022
EPTCS 377, 2023, pp. 65–84, doi:10.4204/EPTCS.377.4

child"), and the requirements that (1) each branch has one left child and one right child, and (2) that every node except the root has one parent. The formalism of graphs also does not account for manipulations: given a binary tree whose leaves are themselves tagged with other binary trees, we might want to collapse this tree of trees into a single tree. While a good starting point, graphs *per se* are not a precise enough formalism to capture these structures and operations.

Church-encodings [22, 5] in polymorphic lambda calculus can precisely express many such structures, and they provide a natural notion of structure-respecting manipulation. For example, the type $\forall X.\ (X \to X \to X) \to (Y \to X) \to X$ encodes exactly binary trees whose leaves are labeled with elements of Y. (Roughly, the two arguments correspond to the two kinds of nodes in binary trees: $X \to X \to X$ corresponds to branch nodes, with two tree-children and one parent; $Y \to X$ corresponds to leaf nodes with one Y-child and one parent.) Further, Church-encodings of structures are themselves functions, corresponding to generalized fold operations: to use a term, you provide one function per constructor, and each occurrence of a constructor in the term is replaced by the corresponding function call. This allows the manipulation of terms in a structure-respecting way. However, standard Church-encodings [5] are over heterogeneous term algebras, but bags, relational database schemas, and syntax trees with variable binding are not term algebras. Finally, these encodings are not canonical, e.g., $\forall X.\ (Y \to X) \to (X \to X \to X) \to X$ also encodes binary trees. As the complexity of encoded structures increases, the number of equivalent encodings may increase combinatorially.

In this work, we leverage the complementary strengths of these two approaches, building a language called Ideograph, where both the terms and the types are graph-structured. By having a calculus, we are able to precisely encode many structures and define structure-respecting operations over them. By having terms that are graphs rather than trees, we are able to capture a richer set of structures. By having types that are graphs, we are able to eliminate many redundant encodings of structures.

We begin by using examples to describe the terms (Section 2.1), operational semantics (Section 2.2), types (Section 2.3), and a well-formedness condition (Section 2.4), followed by the formalism (Section 2.5). We then present representations of several data structures in the language (Section 3) and demonstrate the manipulation of such structures (Section 4). We conclude with discussions of related work (Section 5) and future work (Section 6). For clarity and concision, we omit polymorphism from this presentation.

2 Ideograph

Ideograph is a means of expressing and composing structured graphs. Its terms are "structured" in the sense of having distinct kinds of edges and nodes. Nodes have "ports", and edges connect nodes via these ports. Though we introduce additional constructs to support computation and polymorphism, the core idea is to substitute copies of a graph for certain nodes in another graph. Because the formal definitions of the syntax and semantics have many moving parts and are opaque without context, we begin by stepping through the examples in Figure 1, which demonstrate the key aspects of Ideograph. The formalism is presented in Section 2.5.

2.1 Terms

A term in Ideograph, henceforth an *ideogram*, consists of a set of *boxes* ($\mathcal{B}$, depicted as rounded rectangles e.g. in Figure 1), *nodes* ($\mathcal{N}$, gray circles or rectangles), and *ports* ($\mathcal{P}$, small triangles

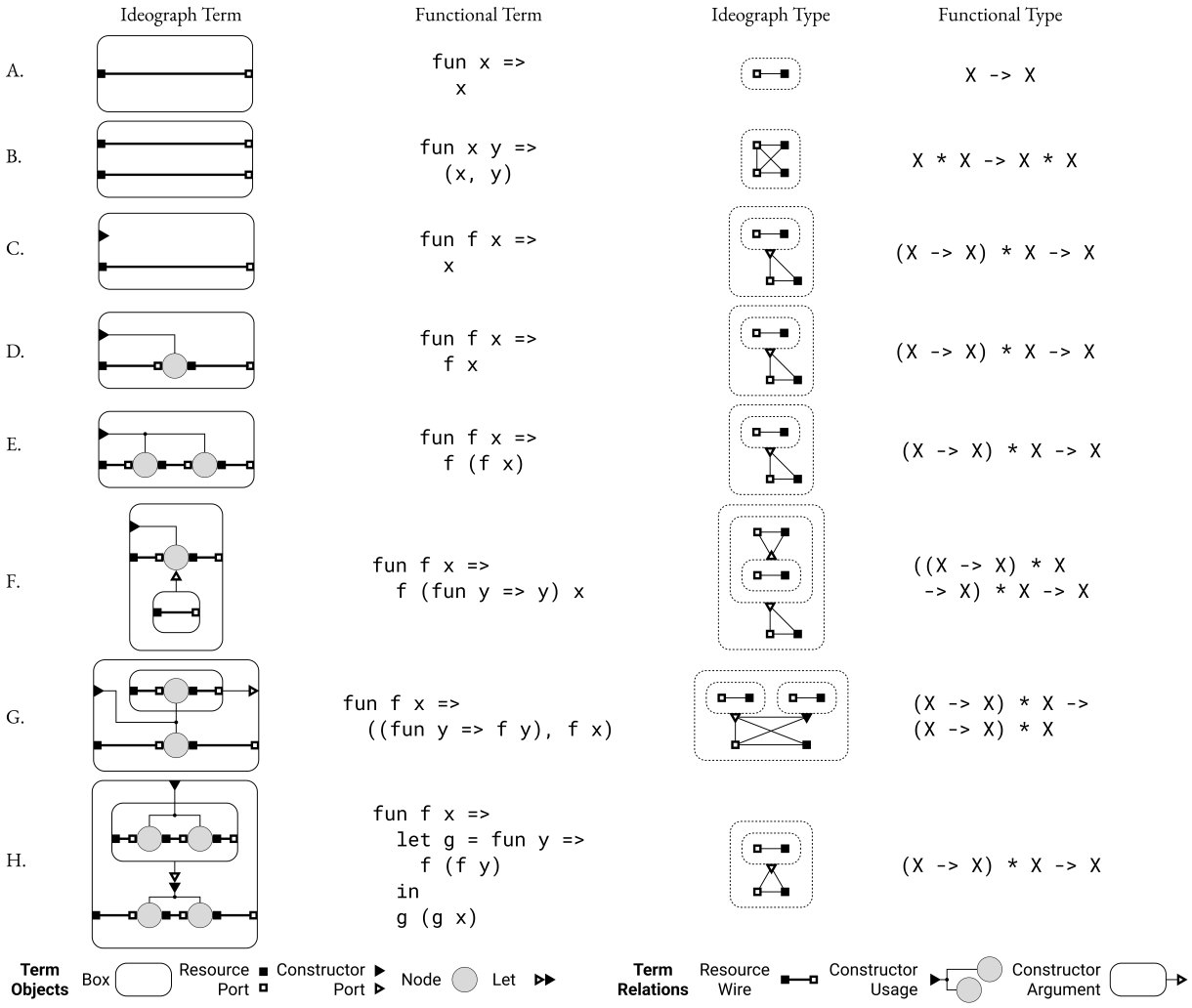


Figure 1: Terms in Ideograph, along with their types, and analogues of each in a generic functional programming language. Though there are multiple Ideograph types that could represent the same functional type, these should provide the right intuition. More precisely, Ideograph's type system is linear in the sense of Girard [11], and here we translated standard function types $X \to Y$ as $!(X \multimap Y)$. This makes function-typed arguments reusable, while other arguments are linear. However, there are other translations [14], and this was a purely expository choice. For simplicity of presentation, we only use one primitive (`X`) in the types, so the types given are not necessarily the principal types of their terms. The formalism allows labeling resource fields with additional primitives.

and squares, hollow or filled), with several relations among these different objects. The boxes, nodes, and ports reside in other boxes (the relation R_R, depicted by nesting), with the boxes forming a tree. Each port is either a *receiver* ($\mathcal{P}_-$, hollow) or a *provider* ($\mathcal{P}_+$, filled) and is for either a *resource* ($\mathcal{P}_R$, square) or a *constructor* ($\mathcal{P}_C$, triangle). Ports are typically *attached* (R_A, depicted by contact) to a node or a box. We will introduce other relations between these objects as they arise in the following examples. Figure 4 and Figure 7 give illustrations annotated with these objects and relations. Terms are also subject to a well-formedness condition that will be discussed in Section 2.4. The comprehensive formalism is deferred to Section 2.5.

Simple functions. Consider the identity function (Example A in Figure 1). As an ideogram, it is a box with two resource ports: one resource provider port (solid square), analogous to the input `x` of the functional analogue; and one resource receiver port (hollow square), analogous to the function output. Because the identity function returns its input as its output, there is a wire between the two resource ports, which is captured by the bijective *resource wiring relation* (R_{WR}, depicted with thick lines) between resource provider and receiver ports.

Example B is slightly more complicated. It has two resource provider ports for the two arguments, `x` and `y`, and two resource receiver ports for the two outputs, the left and right sides of the tuple. The two wires indicate which input gets returned as which output. Because the resource wiring relation is bijective, there are no terms of this type analogous to returning the pairs `(x, x)` or `(y, y)`. This makes resources linear.

Calling functions. Example C is the same as Example A, but with the addition of a single, unused constructor provider port (depicted as a filled triangle, sometimes just called a "constructor"), corresponding to the unused argument `f` with type `X -> X`. This lack of use is allowed because constructors are not linear in the way that resources are.

Examples D and E are more interesting, as they actually use the constructor. Nodes are analogous to function invocations, and so in Example D, we have one node corresponding to the one call to `f`. To capture that the node was constructed by the constructor provider port, the port and node are associated by the *constructor usage relation* (R_{CU}, depicted with a thin, possibly branching line; the branching structure is not meaningful, and exists to improve readability). Each node must be associated with exactly one constructor, but constructors can be associated with any number of nodes. In Example E, the constructor is used to construct two nodes, analogous to the two calls to `f`.

Nodes can have associated ports in the same way that boxes can. In Examples D and E, the nodes each have one resource receiver port, corresponding to the input to the `f` call, and one resource provider port, analogous to the output. In Example D, the wire on the left corresponds to passing the input, `x`, to the call to `f`, and the wire on the right is analogous to returning the output of `f`. In Example E, the wires correspond to passing `x` to the first call to `f`, passing its output to the second call to `f`, and finally returning its output. Note that nodes and boxes have opposite views of provider and receiver: when calling a function (analogous to a node), the function receives the input and provides the output; when defining a function (analogous to a box), the context provides the input and receives the output.

Passing and returning functions. Example F is like Example D, but instead of `f` taking a single argument of type `X`, it also takes an argument of type `X -> X`. This is analogous to the constructor receiver port (hollow triangle) attached to the node. That we are passing the identity function corresponds to the nested box, a copy of Example A, that is connected to the constructor receiver port by the *constructor argument relation* (R_{CA}). This relation is a bijection between constructor receiver ports and boxes (excluding the top-level box). This is the first example with non-trivial box-residence structure: there are two boxes, one (depicted as inner) being the child of the other (depicted as outer).

Example G shows how a function can return a function: the function-typed output is analogous to the constructor receiver port, which, as in Example F, must be connected to a box. Note that the constructor usage relation can sometimes cross box boundaries, analogous to lexical

scoping for functions: in Example G, the constructor provider port corresponding to `f` is used in the inner box. Formally, the constructor usage relation can associate nodes to constructor provider ports in the same box or one that is higher in the residence tree.

Let-bindings. So far we we have only seen values. To have terms that can take operational steps, we also have let-bindings ($\mathcal{D}$, depicted by the contact of a constructor receiver port and a constructor provider port). In Example H, the box connected to the receiver port is analogous to the body of the let-binding, `fun y => f (f y)`, and the two nodes connected to the provider port are analogous to the instances of `g`. Each let-binding must be attached to exactly one constructor receiver port and one constructor provider port. Each port must be associated with exactly one box, node, or let-binding. Each let-binding also has a type, discussed in Section 2.3.

2.2 Operational Semantics

Recall that the core idea of Ideograph is to substitute terms for the nodes of other terms. The let-binding construct connects a box (the binding's body) to some nodes (the occurrences of the binding's bound variable). The single reduction rule of Ideograph is the substitution of the body for the occurrence nodes. (With polymorphism, there is also a type-level let-binding, and there is a second reduction rule for substituting at the type level.)

Consider Figure 2 (i). The let-binding is analogous to the definition of `g`, sequencing two nodes, each analogous to a call to `f`. This let-binding is then used to construct two nodes which are themselves sequenced. Stepping takes the contents of the box (blue) and places a copy of it at each occurrence node (orange). This intuition is depicted in Figure 2 (ii), but requires a bit of clean-up. Each adjacent pair of resource receiver and provider ports is replaced with a wire. The `f` nodes in the body of `g` remain as `f` nodes even after substitution, though we now have four of them. Finally, we erase the let-binding, to get Figure 2 (iii).

Though capturing the core idea of substituting terms for nodes, the previous example does not have constructor ports on the let-binding. Figure 3 does. We again copy the body of the let-binding, `h`, for its occurrence nodes, and then we erase the let-binding and replace each pair of resource receiver and provider ports with a wire. Crucially, the pair of constructor receiver and provider ports becomes a new let-binding. This means that the term can continue stepping. (Indeed, Figure 3 (iii) is the same term as Figure 2 (i).) For a formal definition of the operational semantics and a formalization of this example, see Section 2.5.

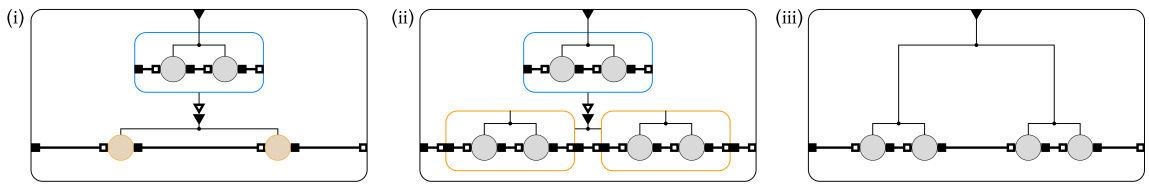


Figure 2: (i) and (iii) depict ideograms and their functional analogues. (i) evaluates to (iii) in one step, by inlining the let-binding. (ii) is not a term, but depicts how inlining is done. Colors indicate analogies.

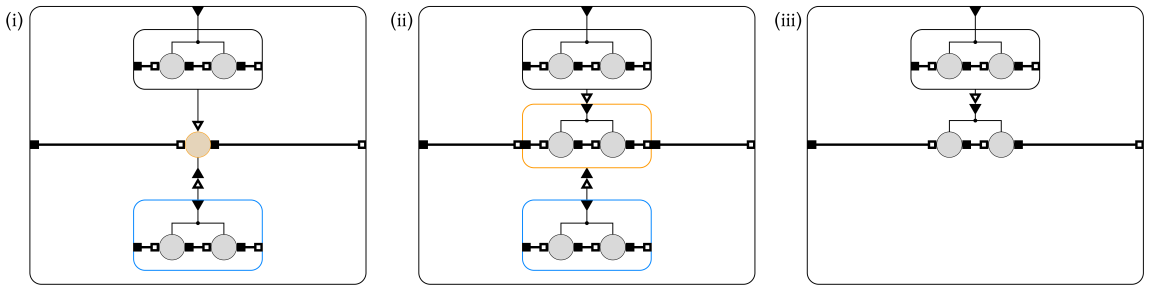


Figure 3: (i) and (iii) depict ideograms and their functional analogues. (i) evaluates to (iii) in one step, by inlining the let-binding. (ii) is not a term, but depicts how inlining is done. Colors indicate analogies. Figure 7 contains a formally annotated version of this example.

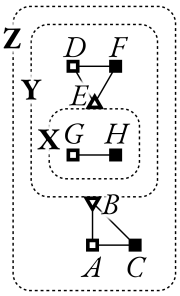
2.3 Types and Correspondences

Doing substitution as outlined above poses a challenge: how do we know the correspondence between the ports on the box and the ports on a node? Figure 2 assumes that the left ports and right ports on the nodes correspond to the left port and right port on the box, respectively. The primary role of types in Ideograph is to make this correspondence precise.

We now define a *type* and, between components of a type ($\mathcal{I}$, $\mathcal{F}$, defined shortly) and components of a term ($\mathcal{B}$, $\mathcal{N}$, $\mathcal{P}$, $\mathcal{D}$), a *correspondence relation*. The counterparts of the ports in a term are the *fields* ($\mathcal{F}$, depicted the same as ports) of a type, and a correspondence relation maps each port to at most one field. In a term, ports are attached to a box or a node, whereas in a type, fields reside in an *interface* ($\mathcal{I}$, depicted as dotted, rounded rectangles), and a correspondence relation maps each box and each node to at most one interface. The residence of fields in an interface, as well as the nesting of interfaces, is captured by the *residence relation* (R_R, depicted by containment), much like it is for terms. Correspondences must be consistent, in that if a box or node corresponds to an interface, then the ports attached to the box or node must correspond bijectively to the fields in the interface. Like ports, fields are either received ($\mathcal{F}_-$) or provided ($\mathcal{F}_+$) and are for either constructors ($\mathcal{F}_C$) or resources ($\mathcal{F}_R$). Constructor and resource ports correspond to constructor and resource fields, respectively. When attached to nodes, receiver and provider ports correspond to receiver and provider fields, respectively. However, when attached to boxes, this is reversed: receivers correspond to providers and providers correspond to receivers. Constructor fields are bijectively associated with interfaces that reside at the same level (R_I, depicted by contact). Correspondences must also be consistent with respect to R_I, in that if a field is associated with an interface, ports corresponding to the field must only be connected to boxes (via R_{CA}) and nodes (via R_{CU}) that correspond to that interface. Finally, in each interface, there is a *connectivity relation* (R_C) between fields, covered in Section 2.4.

Example. In the illustrations, the depiction of correspondences is somewhat implicit, via the positioning of ports (either left, right, top, bottom, top-left, top-right, bottom-left, or bottom-right). In Figure 4, the correspondence (a, A) is depicted by placing both the port and the field on the left of their containers, whereas (b, B) is on the top left and (c, C) is on the right. Since $\mathbf{Y}$ is associated with B and $\mathbf{y}$ is constructed by b, the definition of correspondence relation forces

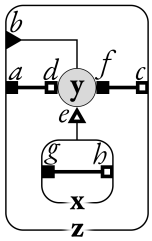
(i)



(ii) $\mathcal{I} = \{\mathbf{Z}, \mathbf{Y}, \mathbf{X}\}$
$\mathcal{F}_{C+} = \varnothing$
$\mathcal{F}_{C-} = \{B, E\}$
$\mathcal{F}_{R+} = \{C, F, H\}$
$\mathcal{F}_{R-} = \{A, D, G\}$
$R_I = \big\{(B, \mathbf{Y}), (E, \mathbf{X})\big\}$

$R_R = \Big\{\big(\mathbf{Z}, \{\mathbf{Y}, A, B, C\}\big),$
$\big(\mathbf{Y}, \{\mathbf{X}, D, E, F\}\big),$
$\big(\mathbf{X}, \{G, H\}\big)\Big\}$
$R_C = \big\{(A, B), (A, C), (B, C),$
$(D, E), (D, F), (E, F), (G, H))\big\}$

(iii)



(iv) $\mathcal{B} = \{\mathbf{z}, \mathbf{x}\}$
$\mathcal{N} = \{\mathbf{y}\}$
$\mathcal{P}_{C+} = \{b\}$
$\mathcal{P}_{C-} = \{e\}$
$\mathcal{P}_{R+} = \{a, f, g\}$
$\mathcal{P}_{R-} = \{c, d, h\}$

$\mathcal{D} = \varnothing$
$\mathcal{T} = \varnothing$
$R_{DI} = \varnothing$
$R_{DC} = \varnothing$
$R_{CA} = \big\{(\mathbf{x}, e)\big\}$
$R_{CU} = \big\{(\mathbf{y}, b)\big\}$

$R_{WC} = \varnothing$
$R_{WR} = \big\{(a, d), (f, c), (g, h)\big\}$
$R_R = \Big\{\big(\mathbf{z}, \{\mathbf{y}, \mathbf{x}, a, b, c, d, e, f\}\big), \big(\mathbf{x}, \{g, h\}\big)\Big\}$
$R_A = \Big\{\big(\mathbf{z}, \{a, b, c\}\big), \big(\mathbf{y}, \{d, e, f\}\big), \big(\mathbf{x}, \{g, h\}\big)\Big\}$

(v) $C = \big\{(\mathbf{z}, \mathbf{Z}), (\mathbf{y}, \mathbf{Y}), (\mathbf{x}, \mathbf{X}), (a, A), (b, B), (c, C), (d, D), (e, E), (f, F), (g, G), (h, H)\big\}$

Figure 4: An annotated illustration (i) and formalization (ii) of the type from Figure 1 F. An annotated illustration (iii) and formalization (iv) of the term from Figure 1 F. The correspondence between them (v), implicitly depicted via field and port positioning. $\{(a, \{b, c\})\}$ is shorthand for $\{(a, b), (a, c)\}$. See Figure 8 for the descriptions of all components of the formalism.

the correspondence $(\mathbf{y}, \mathbf{Y})$. The placement of d on the left of the node and D on the left of the interface depicts the correspondence (d, D), and similarly for (e, E) on the bottom and (f, F) on the right. The correspondence (e, E) forces $(\mathbf{x}, \mathbf{X})$, and then the left and right positionings depict (g, G) and (h, H).

Types internal to terms. Recall the problem of associating ports between the body and occurrences of a let-binding: the solution is to give each let-binding a type, and then, for the body and each occurrence of the let-binding, give a correspondence with the type. In order to do so, terms themselves must contain types, which is accomplished via an *internal type-fragment graph* ($\mathcal{T}$, not depicted), containing the unions of the vertices and edges of zero or more types. In particular, its residence relation ($R_R(\mathcal{T})$, not depicted), may be a forest rather than a tree. The interfaces in $\mathcal{I}(\mathcal{T})$ that are roots of this forest are in bijection (R_{DI}) with the let-bindings. Finally, the *internal correspondence* (R_{DC}, depicted via relative port positioning) is a correspondence relating the body and occurrences of each let-binding with its associated interface. Now each port on an occurrence node is associated with a port on the body box, since they correspond to a shared field in the internal types.

Types external to terms. While the components deriving from let-bindings participate in the internal correspondence, those deriving from the root box of the term do not. Given a type T and a term t, C is an *external correspondence between t and T* if C is a correspondence, if C relates the root of t to the root of T, and if C is disjoint from R_{DC}. When a term is paired with an external correspondence, every box, node, and port (except ports directly attached to let-bindings) corresponds to exactly one interface or field.

Canonicity of terms. A term t at a type T in a functional language translates to, not just an Ideograph term, but the pair of an Ideograph term $(\!|t|\!)_G$ and an external correspondence $(\!|t|\!)_C$ between $(\!|t|\!)_G$ and $(\!|T|\!)$. Consider Figure 1 B. In a traditional functional language, this type has two linear terms: `id := fun x, y => (x, y)` and `swap := fun x, y => (y, x)`. In Ideograph, there is only one term, which is shown, and is equal to $(\!|\texttt{id}|\!)_G = (\!|\texttt{swap}|\!)_G$. However, there are two distinct external correspondences between the term and the type, $(\!|\texttt{id}|\!)_C \neq (\!|\texttt{swap}|\!)_C$. When recursively translating terms, the inner correspondences cancel out, so we have both $(\!|\texttt{id a b}|\!)_G = (\!|\texttt{swap b a}|\!)_G$ and $(\!|\texttt{id a b}|\!)_C = (\!|\texttt{swap b a}|\!)_C$. This quotients out internal argument ordering while allowing `id` and `swap` to be differentiated externally.

2.4 Connectivity Relation and Well-Formedness of Terms

As mentioned previously, types have a connectivity relation, R_C. This is a symmetric, irreflexive relation among the fields residing in each interface. Intuitively, this relation indicates the allowed connections between ports on the "inside" of a node, and thus what might be wired together after substituting for the node. This condition has several benefits: it rules out self-referential let-bindings that could lead to unbounded recursion; it allows us to faithfully represent traditional functions, where the output of a function may not be fed back as an input; and without it, our representations of data structures would not work correctly (see Figure 9 (iv) and Figure 11 (iii)).

Roughly, the connectivity relation differentiates functions and products (more precisely $\otimes$ and $\parr$ in linear logic [11], where linear functions are $A \multimap B := A^\perp \parr B$). While a function may use its input to produce its output (its input and output ports may be connected), a product must produce its left and right sides separately (its left and right ports may not be connected). For a type T, define its *dual*, $T^\perp$, to have the same fields but with, at the top-level, receivers and providers flipped and the complimentary connectivity relation. For types T and S, define $T \sqcup S$ to have the disjoint union of the fields of T and S and the disjoint union of their connectivity relations. Define $T \bowtie S$ to be $T \sqcup S$, but adding connectivity edges between the top-level fields of T and the top-level fields of S. When translating ordinary functional types to Ideograph, $(\!|A \times B|\!)$ becomes $(\!|A|\!) \sqcup (\!|B|\!)$, and $(\!|A \to B|\!)$ becomes $(\!|A|\!)^\perp \bowtie (\!|B|\!)$. See Figure 5 for examples.

For a term to be *well-formed*, the combination of its wiring relation and the connectivity relation from its types must be acyclic, with certain exceptions (see Definition 13). Figure 6 shows well- and ill-formed fragments of terms. There are four fragments, considered with two different types. The type for the top row is analogous to `X -> X`. C is valid, being analogous to an ordinary function call; E is invalid, analogous to a call where the output is fed back as the input; G is valid, analogous to a function that discards its argument by passing it to another

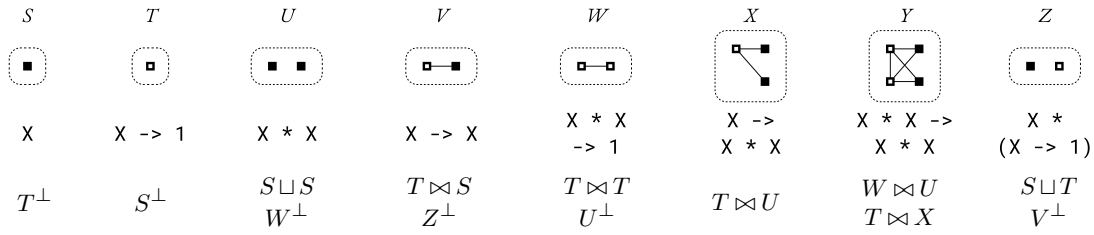


Figure 5: In each column, from top to bottom: a name; a type; a functional analogue; and one or two ways of forming the type from the other types with $\sqcup$, $\bowtie$, and $(\cdot)^\perp$.

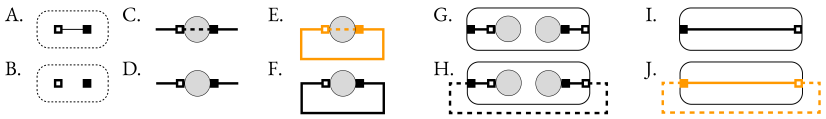
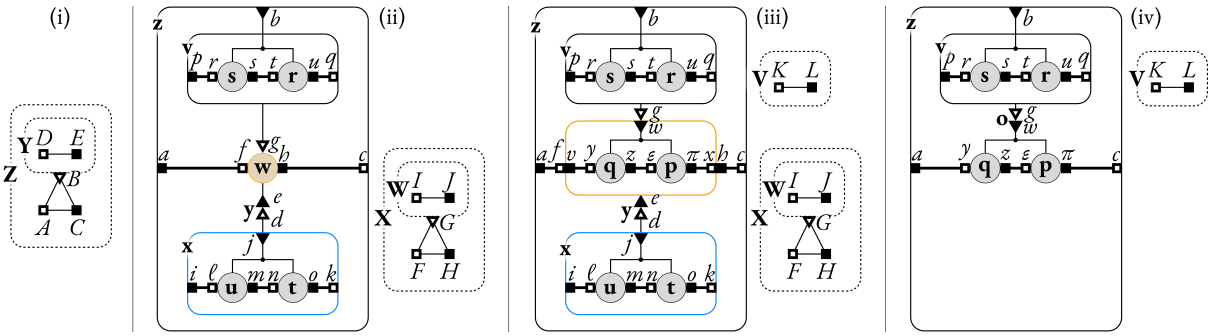


Figure 6: Types (A, B) and fragments of terms (C, D, E, F, G, H, I, J). The interface in A corresponds to the nodes in C and E and to the boxes in G and I. The interface in B corresponds to the nodes in D and F and to the boxes in H and J. The dashed lines are not part of the term, but reflect R_C between the fields of the type, shown between the corresponding ports. Note that because dual types are used for boxes, the dashed lines in G, H, I, and J are the compliment of R_C. E and J are ill-formed terms because of the cycles highlighted in orange.

function and then returning the result of an independent function call; I is valid, analogous to a function that returns its input as its output. The type for the bottom row does not have a perfect analogue in functional programming, but corresponds roughly to `(X -> 1) * X`: a pair of a continuation accepting an `X` and a value of type `X`. D is valid, analogous to using the continuation and the value separately; F is valid, analogous to passing the value to the continuation; H is valid, analogous to constructing the continuation and value with separate function calls; J is invalid, analogous to returning the argument eventually passed to the continuation as the right side of the pair. Both E and J have prohibited cycles.



(v)

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

(vi)

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

Figure 7: An annotated step of the operational semantics at type (i), from term (ii) to term (iv) (shown previously in Figure 3). An illustration of an intermediate step, which is not a term (iii). The partial formalizations of the before term (v) and the after term (vi), and their correspondences C to the type in (i). The omitted pieces of the formalizations are analogous to those in Figure 4. The internal types $\mathcal{T}$, usually not depicted, are shown here. Term (ii) contains an internal interface $\mathbf{X}$, which is the type of the let-binding $\mathbf{y}$. Stepping at $\mathbf{y}$ substitutes the contents of $\mathbf{x}$ (the body of $\mathbf{y}$) for $\mathbf{w}$ (the occurrence of $\mathbf{y}$) and copies $\mathbf{W}$ to $\mathbf{V}$ (shown in (iii)). Finally, the pairs of resource ports f, v and x, h are replaced with wire, while g, w, and $\mathbf{V}$ are attached to a fresh let-binding, $\mathbf{o}$, yielding term (iv).

2.5 Formalism

We now provide a precise formulation of Ideograph. We suggest referring to Figures 4 and 7 to ground definitions as they are introduced.

Definition 1 (fragment graphs)**.** We define *type-fragment graphs* and *term-fragment graphs* in Figure 8. Each consists of several sets of vertices and several edge relations with conditions.

Definition 2 (types and terms)**.** A type-fragment graph is a *type* if R_R has a single root interface. A term-fragment graph is a *term* if R_R has a single root box.

Component	Kind	Description
$\mathcal{I}$	set	*interfaces*, depicted as dotted, rounded rectangles
$\mathcal{F}$	set	*fields*, divided into constructor ($\mathcal{F}_C$, triangle) or resource ($\mathcal{F}_R$, square) and provided ($\mathcal{F}_+$, solid) or received ($\mathcal{F}_-$, hollow)
R_R	$(\mathcal{I} \rightharpoonup \mathcal{I}) \otimes (\mathcal{F} \to \mathcal{I})$, acyclic	*residence*, depicted by containment in interfaces
R_I	$\bigotimes_{i \in \mathcal{I}} \mathcal{F}_C^i \leftrightarrow \mathcal{I}^i$	*constructor-interface* association, depicted by contact between fields and interfaces
R_C	$\bigotimes_{i \in \mathcal{I}}$ cograph on $\mathcal{F}^i$	*connectivity*, depicted with solid lines; *cograph* is Definition 3
$\mathcal{B}$	set	*boxes*, depicted as solid, rounded rectangles
$\mathcal{N}$	set	*nodes*, depicted as gray circles or rectangles
$\mathcal{P}$	set	*ports*, divided into constructor ($\mathcal{P}_C$, triangle) or resource ($\mathcal{P}_R$, square) and provided ($\mathcal{P}_+$, solid) or received ($\mathcal{P}_-$, hollow)
$\mathcal{D}$	set	*let-bindings*, depicted as contact between constructor ports
$\mathcal{T}$	type-fragment graph	*internal type-fragment graph*, not depicted
R_R	$(\mathcal{B} \rightharpoonup \mathcal{B}) \otimes (\mathcal{N} \cup \mathcal{P} \cup \mathcal{D} \to \mathcal{B})$, acyclic	*residence*, depicted by containment in boxes
R_A	$\bigotimes_{b \in \mathcal{B}} \mathcal{P}^b \to (\mathcal{N}^b \cup \mathcal{D}^b \cup \{b\})$	*attachment*, depicted by contact between ports and nodes/boxes
R_{WR}	$\bigotimes_{b \in \mathcal{B}} \mathcal{P}_{R+}^b \leftrightarrow \mathcal{P}_{R-}^b$	*resource wiring*, depicted with thick lines
R_{WC}	$\bigotimes_{b \in \mathcal{B}} \mathcal{P}_{C+}^b \asymp \mathcal{P}_{C-}^b$	*constructor wiring*, not depicted
R_{CA}	$\bigotimes_{b \in \mathcal{B}} \mathcal{B}^b \leftrightarrow \mathcal{P}_{C-}^b$	*constructor argument*, depicted with thin lines
R_{CU}	$\bigotimes_{b \in \mathcal{B}} \mathcal{N}^b \to \bigcup_{b \sqsubseteq b'} \mathcal{P}_{C+}^{b'}$, wire-safe	*constructor usage*, depicted with thin, possibly branching lines; *wire-safe* is Definition 4
R_{DI}	$\mathcal{D} \leftrightarrow$ roots of $\mathcal{I}(\mathcal{T})$	*let-binding typing*, not depicted
R_{DC}	$\bigotimes_{d \in \mathcal{D}}$ correspondence for d	*let-binding correspondence*, depicted by the positioning of ports on boxes and nodes; *correspondence for d* is Definition 6

Figure 8: Components of a type-fragment graph (top) and a term-fragment graph (bottom). $\mathcal{R} \leftrightarrow \mathcal{S}$ is a one-to-one relation, $\mathcal{R} \to \mathcal{S}$ is many-to-one, $\mathcal{R} \rightharpoonup \mathcal{S}$ is many-to-one-or-zero, and $\mathcal{R} \asymp \mathcal{S}$ is many-to-many. $\mathbf{S} \otimes \mathbf{R} = \{S \cup R : S \in \mathbf{S}, R \in \mathbf{R}\}$. $\mathcal{S}^i$ and $\mathcal{S}^b$ denote the subsets of $\mathcal{S}$ residing directly in i and b. $c \sqsubseteq i$ and $c \sqsubseteq b$ mean that c is below i and b in the residence forest. $\mathcal{F}_{C-} = \mathcal{F}_C \cap \mathcal{F}_-$, and likewise for $\mathcal{F}_{C+}$, $\mathcal{F}_{R-}$, $\mathcal{F}_{R+}$, $\mathcal{P}_{C-}$, $\mathcal{P}_{C+}$, $\mathcal{P}_{R-}$, and $\mathcal{P}_{R+}$.

Definition 3 (cographs)**.** The set of *cographs on* $\mathcal{V}$ is the smallest set of symmetric, irreflexive graphs on vertices $\mathcal{V}$ that contains the singleton graphs and is closed under complement and disjoint union. Intuitively, this is the set of formulas on atoms $\mathcal{V}$ that can be formed with conjunction and disjunction, quotienting out associativity and commutativity.

Definition 4 (wire-safety)**.** Assume a constructor usage relation R_{CU}, a constructor wiring relation R_{WC}, and a pair $(n,c) \in R_{CU}$. Let c reside in b_c and n reside in b_n, where $b_n \sqsubseteq b_c$. If $b_n \neq b_c$, let b'_n be the box residing directly in b_c such that $b_n \sqsubseteq b'_n \sqsubset b_c$, and let c' be the constructor receiver port (in b_c) associated with b'_n. R_{CU} is *wire-safe* for R_{WC} if, for all $(n,c) \in R_{CU}$, either $b_n = b_c$ or $(c,c') \in R_{WC}$.

Definition 5 (correspondences)**.** Given a type T and a term t, a relation $C \in (\mathcal{B} \rightharpoonup \mathcal{I}) \otimes (\mathcal{N} \rightharpoonup \mathcal{I}) \otimes (\mathcal{P} \rightharpoonup \mathcal{F})$ is a a *correspondence* if the following hold: (1) If $b \in \mathcal{B}$ (or $n \in \mathcal{N}$) corresponds to $\iota \in \mathcal{I}$, then the correspondence relation is bijective between the ports attached to b (or n) and the fields of ι. (2) If $p \in \mathcal{P}_C$ corresponds to $f \in \mathcal{F}_C$, then the box associated with p corresponds to the interface associated with f. (3) Constructor and resource ports are associated to constructor and resource fields, respectively. (4) If p corresponds to f, their receiver and provider kinds are the same if p is attached to a node and opposite if p is attached to a box.

Definition 6 (let-binding correspondences)**.** A correspondence C is a *correspondence for* $d \in \mathcal{D}$ if the box of d (via R_A and R_{CA}) and nodes of d (via R_A and R_{CU}) correspond to the interface of d (via R_{DI}). Distinct let-binding correspondences must cover disjoint sets of term components.

Definition 7 (external correspondences)**.** Given a type T an a term t, we say that a correspondence relation C is an *external correspondence* between t and T if C relates the root box of t with the root interface of T and C is disjoint from R_{DC}.

Remark. Given a type T, a term t, and an external correspondence C between t and T. Let $C^* \coloneqq C \cup R_{DC}$. Every $n \in \mathcal{N}$ and $b \in \mathcal{B}$ occurs exactly once in C^*. For every $p \in \mathcal{P}$, either it is attached to some $d \in \mathcal{D}$ and does not occur in C^*, or it occurs exactly once in C^*.

Definition 8 (term equality)**.** Given a type T, terms t_1 and t_2, and correspondences C_1 between t_1 and T and C_2 between t_2 and T, we say that (t_1, C_1) is T*-equal* to (t_2, C_2) if, fixing a concrete labeling of vertices to yield $\widehat{T}$, $\widehat{t_1}$, $\widehat{t_2}$, $\widehat{C_1}$, and $\widehat{C_2}$, there exists some relabeling h of the vertices in $\widehat{t_2}$ such that $(\widehat{t_1}, \widehat{C_1}) = (h(\widehat{t_2}), h(\widehat{C_2}))$.

Definition 9 (substitution)**.** Assume a type T, term t, and correspondence C between t and T. Given $b \in \mathcal{B}$ and $n \in \mathcal{N}$, where b resides in some $b_0 \in \mathcal{B}$ and n is at or below b_0 in the residence forest, and given $\iota \in \mathcal{I}(\mathcal{T})$ and correspondences $C_b \subseteq R_{DC}$ between b and ι and $C_n \subseteq R_{DC}$ between n and ι, we define the *substitution of* (b, C_b) *for* (n, C_n) *at* ι to be the result if we: (1) Delete n (from $\mathcal{N}$ and all relations). (2) For each component residing in b, make a fresh copy residing in the box that contained n, also making appropriate copies in R_{DC} and C. (3) For each port p_b attached to b, let p'_b be the fresh copy of p_b, and let $C'_{bp} \subseteq R_{DC}$ be the portion relevant to p'_b. Note that, for each f residing in ι, we now have a p'_b that is fresh and a p_n that used to be attached to n, and that they are a received/provided pair. Let $C_{np} \subseteq C_n$ be the part relevant to p_n. (4) For each resource field f, p'_b was wired to some p''_b and p_n was wired to some p'_n. Erase f, p'_b, and p_n and add (p''_b, p'_n) to the wiring relation. (5) For each constructor field f, create a new let-binding d, and attach p'_b and p_n to it. For the $\iota_f \in \mathcal{I}(\mathcal{T})$ associated with f, make a fresh copy of its subtree in $\mathcal{I}(\mathcal{T})$ and let the root of the copy be ι'_f. Change C'_{bp} and C_{np} to refer to ι'_f. Add (d, ι'_f) to R_{DI}.

Definition 10 (inlining of let-bindings)**.** Assume a type T, term t, and correspondence C between t and T. Given a let-binding $d \in \mathcal{D}$, let the associated interface be ι, the attached constructor receiver port be p_-, the attached constructor provider port be p_+, the argument to p_- be the box b, the set of nodes produced by p_+ be N, the subset of R_{DC} relevant to b be C_b, and for each $n \in N$, the subset of R_{DC} relevant to n be C_n. Define the *inlining of* d to be the result if we: (1) For each $n \in N$, substitute (b, C_b) for (n, C_n) at ι. (2) Delete d, ι, p_-, p_+, and b.

Definition 11 (reduction)**.** Given a type T, terms t_1 and t_2, and correspondences C_1 between t_1 and T and C_2 between t_2 and T, we say that (t_1, C_1) *T-reduces* to (t_2, C_2) if there exists a $d \in \mathcal{D}(t_1)$ such that the result of inlining d in (t_1, C_1) is T-equal to (t_2, C_2).

Definition 12 (descent of components)**.** A *component* is a box, node, port, or let-binding. A component c_1 *is a child of* c_2 if c_1 is attached (related in R_A) to c_2, if c_1 is the constructor argument (related in R_{CA}) of c_2, or if c_1 is a constructor usage (related in R_{CU}) of c_2. For the transitive closure of this relation, c_1 *descends from* c_2. Note that the descent relation forms a forest, separate from the residence forest (R_R), and note that every component descends either from a let-binding or one of the roots of the residence forest.

Definition 13 (well-formedness)**.** Assume a type T, a term t, and an external correspondence C between t and T. Let W be the relation on ports $R_{WR} \cup R_{WC}$. Define the relation F on ports such that $(p_1, p_2) \in F$ if either: (1) p_1 and p_2 are attached to the same $d \in \mathcal{D}$; (2) let c be the nearest common ancestor box or node of p_1 and p_2 in the descent relation, let p_1', p_2' be the ancestors of p_1, p_2 (respectively) attached to c, and let f_1', f_2' be the fields corresponding to p_1', p_2' (respectively); if there is no such c, $(p_1, p_2) \notin F$; otherwise, if c is a node, then $(p_1, p_2) \in F$ if $(f_1', f_2') \in R_C$; if c is a box, then $(p_1, p_2) \in F$ if $(f_1', f_2') \notin R_C$. Now, we say (T, t, C) is *well-formed* if for every cycle taking alternating edges in F and W, there is some pair of ports p_1, p_2 in this cycle such that $(p_1, p_2) \in F$ but the edge (p_1, p_2) is not part of the cycle. (This is directly inspired by the chorded-acyclic R&B-cograph condition from [25], and extended to handle the nesting of interfaces.)

3 Expressing Data

3.1 Binary Trees

Now we will see how Ideograph represents data, using unlabeled binary trees as an example. In polymorphic lambda calculus, they are represented by the type $\forall X.(X \to X \to X) \to (1 \to X) \to X$: the first argument, $X \to X \to X$, is the (arity-2) branch constructor, and the second

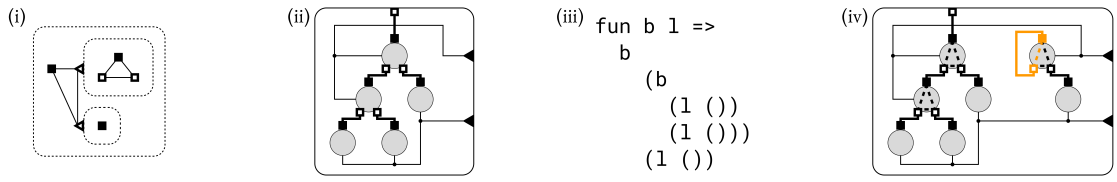


Figure 9: The type of unlabeled binary trees (i). A term of that type (ii). The representation of that term as a Church-encoding in a generic functional language (iii). A term that is ill-formed as an unlabeled binary tree (iv). The dashed lines are not part of the term, but are R_C between the fields of the type, shown between the corresponding ports. The illegal cycle is marked in orange.

argument, $1 \to X$, is the (arity-1) leaf constructor (taking unit, since the trees are unlabeled). Dropping polymorphism, this corresponds to the Ideograph type in Figure 9 (i). The top-right constructor field represents branch nodes, with resource ports for one parent (top), one left child (bottom-left), and one right child (bottom-right); the bottom-right constructor field represents leaf nodes, with a resource port for one parent (top). The remaining resource port (top-left) corresponds to the root of the tree.

Figure 9 (ii) and (iii) represent the same binary tree, with two branch nodes and three leaf nodes. The connectivity relation and well-formedness condition rule out terms like in Figure 9 (iv). Linearity ensures that there is a single tree: additional trees would have no resource port to serve as their root, and such terms would be ruled out by the bijectivity of the resource wiring relation.

3.2 Directed Multigraphs

As an example of a structure that is not a term algebra, and thus lacks a traditional Church-encoding [5], consider directed multigraphs. They are specified in Ideograph by the type in Figure 10 (i), with the top-right field corresponding to "vertices" and the bottom-right field corresponding to "edges". Figure 10 (ii) shows a multigraph with three vertices and three edges and its representation as a term. Because vertices may be associated with any number of edges, vertex nodes have a constructor port rather than a resource port. Edges have two ports, for receiving constructors from their source and target vertices. Here, constructor provider ports are shown directly connected to constructor receiver ports, which is not formally allowed—we abuse notation for clarity, and mean that the receiver port is attached to a box which contains a single node constructed by the provider port. Figure 10 (iii) shows a similar multigraph with three vertices, but with six edges. There is a related type for representing bags, which are essentially multigraphs without edges.

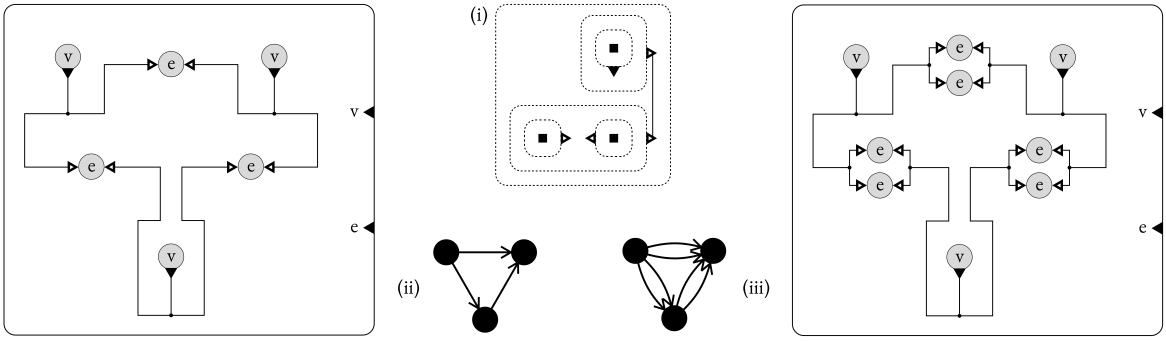


Figure 10: The type of directed multigraphs (i). Two terms and the directed multigraphs they represent, (ii) and (iii). To improve readability, the constructor usage relation is depicted with the labels "v" and "e" rather than lines.

3.3 Untyped Lambda Calculus

Closed terms in untyped lambda calculus also do not form a term algebra. They have three kinds of nodes—application, abstraction, and variable—but every variable node must somehow be associated with an abstraction above it.

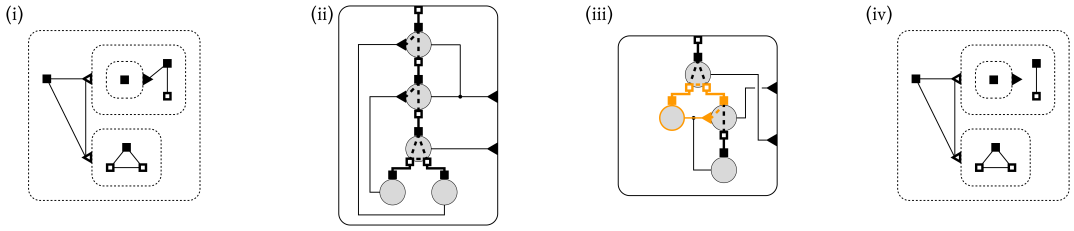


Figure 11: The type of closed terms in untyped lambda calculus (i). A term representing "$\lambda x.\lambda y.\, y\ x$" (ii). A term corresponding to "$x\ (\lambda x.x)$" (iii). Another type, differing from (i) by a single connectivity edge (iv). The connectivity relation from (i) is overlaid with dashed lines on the corresponding ports in (ii) and (iii). The cycle marked in orange makes the external correspondence between (i) and (iii) ill-formed. There is a well-formed external correspondence between (iii) and (iv).

Figure 11 (i) shows their type in Ideograph, with the bottom-right constructor field representing application, having resource fields for a parent (top), a left child (bottom-left), and a right child (bottom-right), and the top-right constructor field representing abstraction, having resource fields for a parent (top) and a child (bottom), and a constructor port for "variable nodes referring to this abstraction" (left). Rather than starting with a single constructor for "variable" nodes, each time an abstraction node is constructed, a new "variable" constructor appears. The connectivity relation on the interface of abstractions prevents variables from occurring above their binder, as in Figure 11 (iii): the edge between the "variable" constructor port and the "parent" port prohibits this, while the lack of edge between the "variable" constructor port and the "child" port allow variables to occur in the body of an abstraction. In contrast, Figure 11 (iv) lacks the variable-parent edge and thus does admit this term.

This representation is closely related to parametric higher-order abstract syntax (PHOAS) [28, 6], which leverages parametric polymorphism to represent variable binding. Computation over our representation, like PHOAS, respects the binding structure of terms, allowing the implementation of single-step beta-reduction and providing capture-avoiding substitution for free.

4 Manipulating Data

Now we show how to manipulate such data structures. The lack of polymorphism in this simplified presentation forces a simple example, since it is not clear how to write many functions over Church-encodings without instantiating the universal quantifier with complex types. Though the full version of Ideograph can represent much richer functions, the following example should

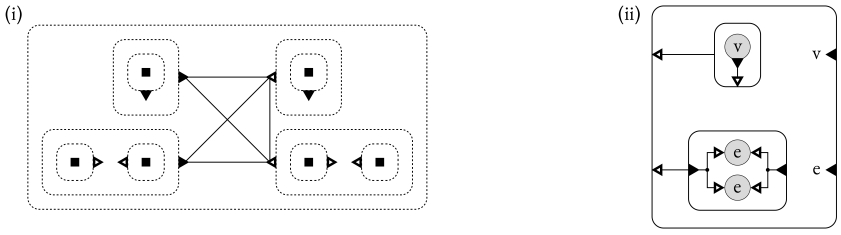


Figure 12: The type of functions from directed multigraphs to directed multigraphs (i). The term of that type that replaces each edge in the input with a pair of edges (ii).

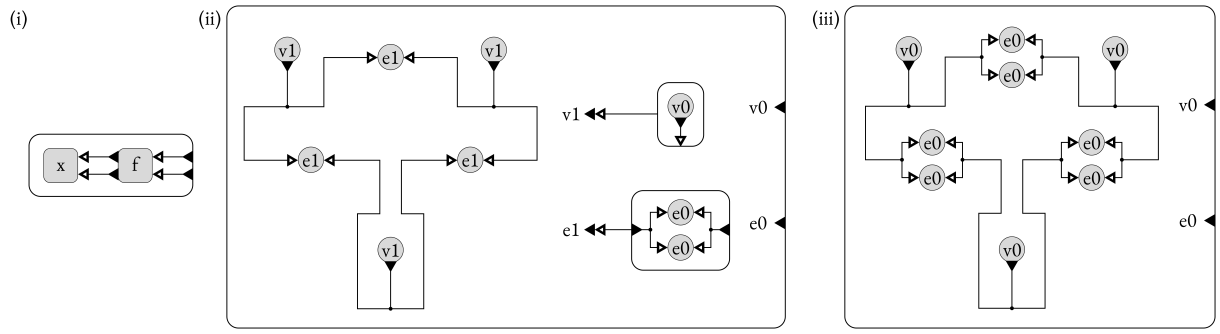


Figure 13: The term that passes an instance of "x" to a call to "f" (i). The result of inlining in (i) the definitions of "x" and "f" (ii). The result of inlining in (ii) the let-bindings "v1" and "e1" (iii). "x" is a let-binding whose body is the directed multigraph in Figure 10 (ii), and "f" is a let-binding whose body is the function in Figure 12 (ii). (iii) is the term from Figure 10 (iii).

provide the right intuition.

Recall the representation of directed multigraphs from Section 3.2. The term in Figure 12 (ii) behaves like a function that takes a directed multigraph as input and doubles each edge. Figure 13 depicts the evaluation of this function on the directed multigraph with three vertices and three edges from Figure 10 (ii).

We emphasize that, in most programming languages, manipulating a graph involves manipulating a structure with labeled nodes—be it an adjacency matrix or a list of pairs of node indices—which makes it possible to write functions that are dependent on the labeling, and thus not truly functions over graphs. Ideograph cannot do that: it merely replaces each node of a structure (in this case both vertices and edges are nodes) with some pattern, as with Church-encodings; in this case, each vertex is replaced by a single vertex, and each edge by two edges of the same orientation.

Unfortunately, this is conservative: there are legitimate functions on graphs that we cannot express, including the function that returns the number of vertices as a Church-numeral. Specifically, Ideograph has two distinct types that resemble the natural numbers, which are roughly "lists of units" and "bags of units"; we can write the function that counts the number of vertices as a bag of units, but we cannot write the function that converts a bag of units to a list of units, even though it would be sound. Though we could add a primitive function to accomplish this, we might wish to define it internally. Characterizing and enlarging the set of functions that can be represented is left for future work.

5 Related Work

Linear logic. Ideograph is closely related to linear logic [11]. Most presentations of linear logic use the rules of "contraction", "weaking", and "dereliction" for exponentials, but Andreoli's equivalent dyadic system [3] instead uses a rule called "adsorption", which is very reminiscent of our nodes and our constructor usage relation. An obvious difference is that our propositions are graphs, not trees, allowing us to quotient out certain type equivalences, like the ordering of products. The type equivalences that we quotient out are similar to the *provable type isomorphisms* for intuitionistic type systems [8]. This leads us to conjecture that Ideograph is

polymorphic (second-order propositional) multiplicative exponential linear logic with the MIX rule, but with these type isomorphisms quotiented out.

Proof nets and interaction nets. Our work is closely related to proof nets [11], and, in particular, their extension, interaction nets [15]. There are two key differences between our work and interaction nets: (1) In interaction nets, each symbol (roughly our "node") has a *principal* port, which is used in reduction; in our work, nodes do not have privileged ports, and reduction proceeds exclusively by substituting definitions (of types or terms) for their occurrences. (2) In interaction nets, the set of symbols and their associated ports must be fixed ahead of time; in our work, the symbol set is not fixed, with occurrences of symbols potentially adding fresh symbols to the set.

There is work on representing lambda calculus terms using interaction nets [19, 18, 21]. This work uses explicit "duplication" and "erasure" symbols, whereas exponentials ("constructors" in our terminology) are a central piece of our formalism. A key advantage of that work is improved reduction performance on some benchmarks, facilitated by the sharing of subterms [18]. We hope to evaluate our system on their benchmark in future work.

There are versions of both proof nets [11] and interaction nets [16, 17] that represent exponentials with "boxes", and we expect these to be closely related to Ideograph, though our types are graphs rather than trees.

Functional programming. There are several key differences between Ideograph and more traditional functional programming languages like OCaml, Haskell, and Rust. (1) Ideograph does not have primitive inductive datatypes, instead using an analogue of Church-encodings. (2) Ideograph is both pure and strongly-normalizing and does not prescribe an evaluation order. (3) Ideograph is linear in the sense of Girard [11], whereas Rust and Linear Haskell lack exponentials, Linear Haskell has separate non-linear types, and Rust is affine. (4) Most languages have functions implicitly return a single value, but boxes ("functions") in Ideograph explicitly name their zero or more outputs, similar to out-parameters in C and similar languages. (5) Ideograph has a natural interpretation as graph substitution, even if a textual formalism were preferred for writing programs.

Polymorphic lambda calculus. There are three key differences between (polymorphic) Ideograph and polymorphic lambda calculus (System F): Ideograph is conjectured to be a canonical version of polymorphic multiplicative exponential linear logic (PMELL) with the MIX rule; PMELL is the classical counterpart to polymorphic intuitionistic linear logic (PILL); and PILL is the linear counterpart to System F. The presentation here is not polymorphic, and so corresponds to intuitionistic linear logic and the simply-typed lambda calculus.

In terms of ability to express data types, we expect Ideograph and PMELL to be the same. However, the canonicity of Ideograph means that e.g. for a directed multigraph g, where in PMELL there is a different (fully-normalized) term for each labeling of the vertices and edges of g, in Ideograph there is a unique (fully normalized) term representing g. We are unsure of how linearity and classicality affect the ability to express data (i.e. the set of types and their fully-normalized terms). Linearity and classicality have established effects on the computational behavior of languages: linear calculi are often able to explicitly distinguish call-by-value and call-by-name [14], and classicality allows the expression of constructs like call/cc [13]. Data structures that form heterogeneous term algebras can be procedurally Church-encoded into

System F types [5], but structures like directed multigraphs and lambda calculus terms are not term algebras.

Graph representations of programming languages. There is work representing existing programming languages, in particular lambda calculus, as graphs [10, 26, 12]. A key difference of our work is that we are not trying to represent existing programming languages for the purposes of, e.g. optimizing compilation. Rather we want to represent data structures (of which syntax trees happen to be one) and pure computations over them. As a result, we are not concerned with effects or evaluation order, with which much of this work contends.

Graph representations of types. There is work on representing formulas in multiplicative linear logic as undirected graphs [2, 25]. Our type system is closely related to these when we do not use exponentials or second-order propositional quantifiers. (Note that we adopt the edge convention opposite of theirs for $\otimes$ and $\parr$.)

Representations of graphs. There is work representing graph structures in pure functional programming languages via recursive binders [20]. While our system is linearly typed and theirs is not, we expect them to be closely related and hope to pursue the connection in future work.

Graph programming languages. There are several graph programming languages, including GROOVE [24], GP-2 [23], and LMNtal [27]. All such systems we are aware of have a notion of a rewrite rule, which matches a subgraph and replaces it with some other subgraph. In contrast, our system has only one reduction rule, which is analogous to beta reduction. LMNtal lacks a type system, whereas types are a core part of Ideograph. HyperLMNtal [29], which extends LMNtal with hyperedges, has been used to encode lambda calculus terms. In contrast to our use of constructors, they connect a binder to all of its variable occurrences via a single hyperedge. Ideograph is a variant of "labeled port graphs" [9], a formalism where edges connect to nodes at "ports", which has been used to represent programs.

Parametric higher-order abstract syntax. There is work on encoding syntax with variable binding using functions in the meta-language. In particular, [28] and [6] leverage parametric polymorphism to encode exactly the closed terms in several lambda calculi and allow only structure-respecting operations. Their encodings, when translated into the types of Ideograph, correspond very closely to the type we presented in Section 3.3. However, Ideograph can represent languages where variables must be used exactly once, and parametric higher-order abstract syntax cannot. Moreover, our goal is to represent structures beyond just syntax.

6 Future Work

There are several avenues for future work. We must first prove standard properties about Ideograph, including subject reduction and strong normalization. We would then like to prove that term-equality is graph isomorphism-complete, to characterize the structures that can be represented by the types, and to formally establish the connection to linear logic. Finally, we plan to develop an implementation, which we expect to be fairly straightforward due to the simplicity of the operational semantics.

Acknowledgements

We would like to thank Ian Mackie, Kazunori Ueda, and two anonymous reviewers for their valuable feedback, as well as Lawrence Dunn, Harrison Goldstein, Eleftherios Ioannidis, Nick Rioux, and Lucas Silver for reading early drafts. This work is funded in part by NSF Awards CCF-1910769 and CCF-1917852.

References

[1] Samson Abramsky (1993): *Computational interpretations of linear logic*. *Theoretical Computer Science* 111(1), pp. 3–57, doi:10.1016/0304-3975(93)90181-R.

[2] Matteo Acclavio, Ross Horne & Lutz Straßburger (2020): *Logic Beyond Formulas: A Proof System on Graphs*. In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, Association for Computing Machinery, New York, NY, USA, p. 38–52, doi:10.1145/3373718.3394763.

[3] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. *Journal of Logic and Computation* 2(3), pp. 297–347, 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 & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The PoplMark Challenge*. In: *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, Springer-Verlag, Berlin, Heidelberg, p. 50–65, doi:10.1007/11541868_4.

[5] Corrado Böhm & Alessandro Berarducci (1985): *Automatic synthesis of typed Λ-programs on term algebras*. *Theoretical Computer Science* 39, pp. 135–154, doi:10.1016/0304-3975(85)90135-5. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[6] Adam Chlipala (2008): *Parametric Higher-Order Abstract Syntax for Mechanized Semantics*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, Association for Computing Machinery, New York, NY, USA, p. 143–156, doi:10.1145/1411204.1411226.

[7] N.G de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. *Indagationes Mathematicae (Proceedings)* 75(5), pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[8] R. Dicosmo (1995): *Second Order Isomorphic Types: A Proof Theoretic Study on Second Order λ-Calculus with Surjective Pairing and Terminal Object*. *Information and Computation* 119(2), pp. 176–201, doi:10.1006/inco.1995.1085.

[9] Maribel Fernández, Hélène Kirchner & Bruno Pinaud (2018): *Labelled Port Graph – A Formal Structure for Models and Computations*. *Electronic Notes in Theoretical Computer Science* 338, pp. 3–21, doi:10.1016/j.entcs.2018.10.002. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[10] Dan R. Ghica, Koko Muroya & Todd Waugh Ambridge (2019): *A robust graph-based approach to observational equivalence*, doi:10.48550/ARXIV.1907.01257.

[11] Jean-Yves Girard (1987): *Linear logic*. *Theoretical Computer Science* 50(1), pp. 1–101, doi:10.1016/0304-3975(87)90045-4.

[12] Clemens Grabmayer (2018): *Modeling Terms by Graphs with Structure Constraints (Two Illustrations)*. In Maribel Fernández & Ian Mackie, editors: *Proceedings Tenth International Workshop on Computing with Terms and Graphs, TERMGRAPH@FSCD 2018, Oxford, UK, 7th July 2018*, *EPTCS* 288, pp. 1–13, doi:10.4204/EPTCS.288.1.

[13] Timothy G. Griffin (1989): *A Formulae-as-Type Notion of Control*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, p. 47–58, doi:10.1145/96709.96714.

[14] Giulio Guerrieri & Giulio Manzonetto (2018): *The Bang Calculus and the Two Girard's Translations*. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva & Lorenzo Tortora de Falco, editors: *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018*, *EPTCS* 292, pp. 15–30, doi:10.4204/EPTCS.292.2.

[15] Yves Lafont (1989): *Interaction Nets*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, p. 95–108, doi:10.1145/96709.96718.

[16] Yves Lafont (1995): *From proof nets to interaction nets*, p. 225–248. London Mathematical Society Lecture Note Series, Cambridge University Press, doi:10.1017/CBO9780511629150.012.

[17] Ian Mackie (2000): *Interaction nets for linear logic*. *Theoretical Computer Science* 247(1), pp. 83–140, doi:10.1016/S0304-3975(00)00198-5.

[18] Ian Mackie (2011): *An Interaction Net Implementation of Closed Reduction*. In Sven-Bodo Scholz & Olaf Chitil, editors: *Implementation and Application of Functional Languages*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 43–59, doi:10.1007/978-3-642-24452-0_3.

[19] Ian Craig Mackie (1994): *The Geometry of Implementation*. PhD thesis, Imperial College of Science, Technology and Medicine, doi:10.25560/46072.

[20] Bruno C.d.S. Oliveira & William R. Cook (2012): *Functional Programming with Structured Graphs*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, Association for Computing Machinery, New York, NY, USA, p. 77–88, doi:10.1145/2364527.2364541.

[21] Vincent van Oostrom, Kees Jan van de Looij & Marijn Zwitserlood (2004): *Lambdascope Another optimal implementation of the lambda-calculus*.

[22] Benjamin C. Pierce (2002): *Types and Programming Languages*, 1st edition. The MIT Press.

[23] Detlef Plump (2011): *The Design of GP 2*. In Santiago Escobar, editor: *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011*, *EPTCS* 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[24] Arend Rensink (2004): *The GROOVE Simulator: A Tool for State Space Generation*. In John L. Pfaltz, Manfred Nagl & Boris Böhlen, editors: *Applications of Graph Transformations with Industrial Relevance*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 479–485, doi:10.1007/978-3-540-25959-6_40.

[25] Christian Retoré (2003): *Handsome proof-nets: perfect matchings and cographs*. *Theoretical Computer Science* 294(3), pp. 473–488, doi:10.1016/S0304-3975(01)00175-X. Linear Logic.

[26] Ralf Schweimeier & Alan Jeffrey (1999): *A Categorical and Graphical Treatment of Closure Conversion*. *Electronic Notes in Theoretical Computer Science* 20, pp. 481–511, doi:10.1016/S1571-0661(04)80090-2. MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference.

[27] Kazunori Ueda (2009): *LMNtal as a hierarchical logic programming language*. *Theoretical Computer Science* 410(46), pp. 4784–4800, doi:10.1016/j.tcs.2009.07.043. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.

[28] Geoffrey Washburn & Stephanie Weirich (2003): *Boxes Go Bananas: Encoding Higher-Order Abstract Syntax with Parametric Polymorphism*. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, Association for Computing Machinery, New York, NY, USA, p. 249–262, doi:10.1145/944705.944728.

[29] Alimujiang Yasen & Kazunori Ueda (2021): *Revisiting Graph Types in HyperLMNtal: A Modeling Language for Hypergraph Rewriting.* *IEEE Access* 9, pp. 133449–133460, doi:10.1109/ACCESS.2021.3112903.