Ideograph: A Language for Expressing and Manipulating Structured Data

(Work in Progress)

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 describes kinds of structures, such as natural numbers, lists, multisets, binary trees, syntax trees with variable binding, directed acyclic graphs, and relational databases. Fully normalized terms of a type correspond exactly to members of the corresponding structure, analogous to a Church encoding. Non-normal terms encode alternate representations of their fully normalized forms. In this paper, we first illustrate the correspondence between terms in our language and standard Church encodings, and then we exhibit the type of closed terms in untyped lambda calculus.

1 Introduction

Structured data is ubiquitous: lists, trees, dependency 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 [3], 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 [2].

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 truly is a function over bags rather than lists. For manipulating syntax trees, we can separately prove that our substitution operation is capture-avoiding. However, this is labor-intensive, and it must be re-done for each new structure and operation. We want a general formalism for representing and manipulating a rich class of structured data.

Submitted to:
SOS 2007

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 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 [16] 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 in polymorphic lambda calculus are not able to represent many non-tree structured types, like bags, relational database schemas, and syntax trees with variable binding, all of which require non-trivial equivalences or constraints on the structure. Finally, these encodings are not canonical, e.g., $\forall X.\ (Y \to X) \to (X \to X \to X) \to X$ (with the order of arguments reversed) 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 polymorphic language 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 the same structures.

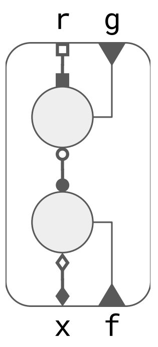
We begin in Section 2 by drawing the correspondence between Church encodings in more traditional systems and the terms of our language. We then present our type system and an encoding of closed terms in untyped lambda calculus, which we believe to be novel. We conclude in with discussions of related work (Section 3) and future work (Section 4). For interested readers, the formal definitions of types and terms can be found in Appendices A and B.

2 Ideograph

Ideograph is a means of expressing and composing structured graphs. Our terms (henceforth ideograms) are "structured" in the sense of having distinct kinds of edges and nodes, and having edges attach to the "ports" of nodes, in the same vein as "port graphs" [5]. Though we introduce additional constructs in order to support computation and polymorphism, the core idea is to substitute (open) port graphs for nodes of other port graphs.

2.1 Notation

Consider the linearly-typed[1] pseudocode in Figure 1: t_0 composes its input functions f and g. The diagram depicts the corresponding ideogram: `f` and `g` are each called once, `x` is passed to the `f` call, `f`'s output passed to the `g` call, and `g`'s output is finally returned (`r`).



```
let $t_0$
  $(x : D)$
  $(f : !(D \multimap C))$
  $(g : !(C \multimap S))$
  $: S$
$= g\ (f\ x)$
```

Figure 1: A simple ideogram and the corresponding pseudocode. The textual labels are for reference, and not part of the term.

The gray circles correspond to function calls, and we call them *gates*. The black triangles correspond to functions, and are linked to their calls by thin lines. We call these triangles *factories*. (Calling the function f is analogous to the factory `f` "producing" a gate.).[2] The thick lines we call *wires*, and they indicate how data flows between function calls (e.g. that x is passed as input to f, that the output of f is passed as input to g, and that the output of g is returned).[3]

Both gates and the boundaries of ideograms have *ports*, corresponding to their inputs and outputs. The `f` gate has one hollow diamond port and one solid circle port, the `g` gate has one hollow circle port and one solid square port, and the term boundary has one hollow square (`r`), one solid diamond (`x`), and two factory (`f` and `g`) ports. Ports are depicted as hollow or solid according to whether something is an input or an output, respectively. (Keep in mind that this perspective is flipped when defining a function: x, f, and g are all parameters of t, but inside the body of t they are given, not taken.)

There are three kinds of ports: resource ports (e.g. `x` and `r`), factory ports (e.g. `f` and `g`), and quantifier ports (to be seen shortly). Resource ports correspond to linearly typed inputs and outputs (e.g. x and the return value of t_0) and are depicted with small symbols corresponding to their type (diamond for D, circle for C, and square for S). Factory ports correspond to inputs and outputs that are functions.[2] The ports of gates are determined by the type of their factory: the gate produced by `f` has one hollow diamond and one solid circle port because $!(D \multimap C)$, the type of f, has one D input and one C output.

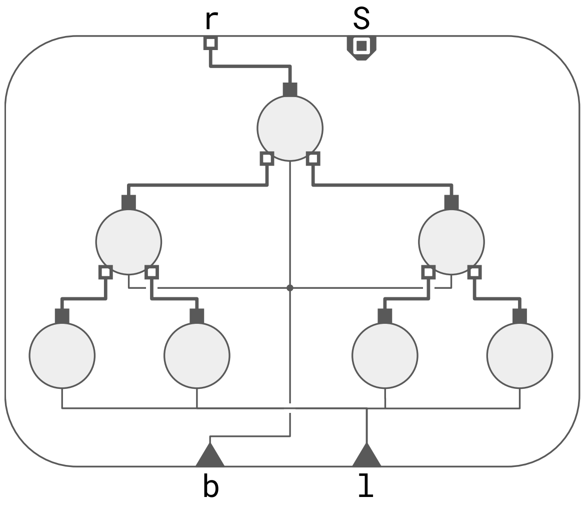
2.2 Representing Binary Trees

Now we will see how data can be represented, using unlabeled binary trees as an example. In System F, they correspond to the type $\forall S.S \to (S \to S \to S) \to S$: the first argument, S, is the (arity-0) leaf constructor and the second argument, $(S \to S \to S)$, is the (arity-2) branch constructor. While there are several translations from intuitionstic to linear types, here we use $\forall S.!(S) \multimap !(S \multimap S \multimap S) \multimap S$.

[1] If you are unfamiliar with linear types, $x : D$ is a D that must be used exactly once, whereas $x : !D$ produces a new D each time it is used. $D \multimap C$ is the type of linear functions from D to C, where the input D is consumed by the function and the output C must be used exactly once.

[2] Though this is intuitively correct, more precisely, factories correspond to things of exponential type and gates correspond to contraction-then-dereliction.

[3] By treating function calls as nodes in a graph, we are implicitly abandoning a fixed evaluation strategy. However, we are concerned with pure, terminating computations, so this is okay.



let t_1 $(S : *)$ $(\ell : !S)$ $(b : !(S \multimap S \multimap S)) : S :=$
  $b\ (b\ \ell\ \ell)\ (b\ \ell\ \ell)$

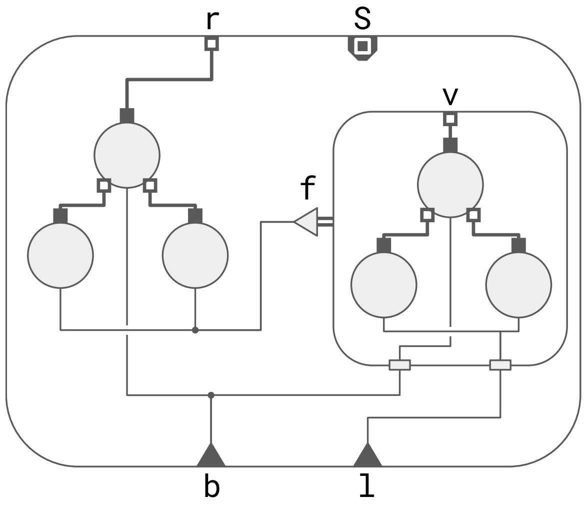
Figure 2: A Church-encoded unlabeled binary tree, as an ideogram and as pseudocode.

In Figure 2, b (resp. ℓ) is called 3 (resp. 4) times, corresponding to the 3 (resp. 4) gates produced by the `b` (resp. `l`) factory. Because the type of b is $!(S \multimap S \multimap S)$, the gates produced by `b` have two square input ports and one square output port. Because the type of ℓ is $!(S)$, the gates produced by `l` have no input ports and one square output port. `S` is a quantifier port, corresponding to the type parameter $(S : *)$. Quantifier ports are depicted as half-octagons, containing the symbol corresponding to their bound variable (in this case, a square). Because S does not occur in the term t_1, but is used in typing, the `S` port is not directly connected to anything, but squares do appear as the types of resource ports.

2.3 Operational Semantics

So far we have only seen normal terms, which cannot take computational steps. Figure 3 depicts a term which can. In the pseudocode, we have a let-binding that defines f, and computation can proceed by inlining the definition of f for each of its two occurrences. In the ideogram, the factory below the `f` label corresponds to the variable f, and the inner term in the box to the right corresponds to the body of the let-binding. Computation proceeds by replacing each gate produced by the `f` factory with a copy of the the `f` inner term, resulting in the ideogram from Figure 2. Despite its simplicity, this provides immense computational power, directly analogous to beta reduction in lambda calculus.

The small gray rectangles on the bottom edge of the inner term represent the closure: the thin lines from the `b` and `l` factories to these rectangles indicate that the factories are captured in the closure; the thin lines from the rectangles to gates inside the inner term indicate that the gates were produced by the closure's factories. The closure is represented more explicitly here than in the pseudocode, but the gates in the inner term correspond to the one b and two ℓ calls



let t_2 $(S : *)$ $(\ell : !S)$ $(b : !(S \multimap S \multimap S)) : S :=$
  let $f : S := b\ \ell\ \ell$ in
  $b\ f\ f$

Figure 3: A non-normal term which evaluates to the term in Figure 2.

in the body of f.

2.4 Ideograph Types

Up to this point, we have restricted ourselves to examples that can be expressed in intuitionistic linear type systems. We will now transition to the types of Ideograph and demonstrate its expressive power by representing syntax trees with variable binding.

Here we use use metavariables A, B for types and X, Y for type variables. Ideograph types are cographs[4] whose nodes are labeled with exactly one of $\underline{X}$, $\overline{X}$, $!A$, $¡A$, $\exists X$, and $\forall X$. Note that $!A, ¡A$ allow for recursive structure, where nodes in types are themselves labeled with types. $\exists X$ and $\forall X$ are binding occurrences of X. $\underline{X}$ and $\overline{X}$ must refer to some X bound by another node in the type or in some outer scope. By $\{N\}$ we mean the type that is the singleton graph where the node is labeled with N. Given types A and B, we define $A \sqcup B$ (resp. $A \bowtie B$) to be the disjoint union of A and B with no edges between $u \in A$ and $v \in B$ (resp. with an edge between each $u \in A$ and every $v \in B$). We follow the usual Barendregt conventions that implicitly rename variables to avoid capture.

The following is a translation from formulas in polymorphic second-order multiplicative linear logic[5] (which with $A \multimap B := A^\perp \parr B$ is a superset of the intuitionistic linear types we have seen), to Ideograph types. We assume negation only occurs at variables, and by $A^\perp$ we

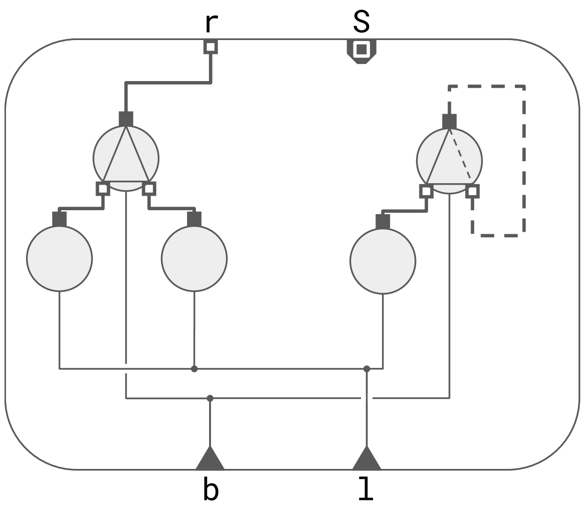
[4]The set of cographs is the smallest set of irreflexive, undirected graphs containing the empty and singleton graphs and closed under disjoint union and complimentation [19].

[5]The fragment of Girard's linear logic [7] consisting of propositional quantifiers, $\otimes$, $\parr$, !, and ?.

mean to propagate negation to the variables of A via de Morgan laws.

$$\begin{array}{llll}
\llbracket A \otimes B \rrbracket := \llbracket A \rrbracket \sqcup \llbracket B \rrbracket & \llbracket \exists X.A \rrbracket := \{\exists X\} \sqcup \llbracket A \rrbracket & \llbracket !A \rrbracket := \{!\llbracket A \rrbracket\} & \llbracket X \rrbracket := \{\underline{X}\} \\
\llbracket A \parr B \rrbracket := \llbracket A \rrbracket \bowtie \llbracket B \rrbracket & \llbracket \forall X.A \rrbracket := \{\forall X\} \bowtie \llbracket A \rrbracket & \llbracket ?A \rrbracket = \{\text{¡}\llbracket A^{\perp} \rrbracket\} & \llbracket X^{\perp} \rrbracket := \{\overline{X}\}
\end{array}$$

Recall that the intuitionistic type of unlabeled binary trees was $\forall S.!(S) \multimap !(S \multimap S \multimap S) \multimap S$, which translates to the Ideograph type in Figure [4]. This corresponds closely to the diagram: gates produced by `b` have two hollow square ports and one solid square port because the type of `b` has two "$\overline{S}$"s and one "$\underline{S}$". Gates produced by `l` have one solid square port because the type of `l` has one "$\underline{S}$".



$$\underbrace{\{\forall S\}}_{\texttt{S}} \bowtie \underbrace{\{\text{¡}\{\underline{S}\}\}}_{\texttt{l}} \bowtie \underbrace{\{\text{¡}(\{\overline{S}\} \bowtie \{\overline{S}\} \bowtie \{\underline{S}\})\}}_{\texttt{b}} \bowtie \underbrace{\{\underline{S}\}}_{\texttt{r}}$$

Figure 4: The type of unlabeled binary trees (bottom) and an invalid term of that type (top). The labels on fragments of the type correspond to the labels on the ports in the ideogram. The lines on top of the gates produced by `b` are not part of the term, but are shown to reflect the edges in the type of `b`. The dashed lines identify an illegal cycle in the term.

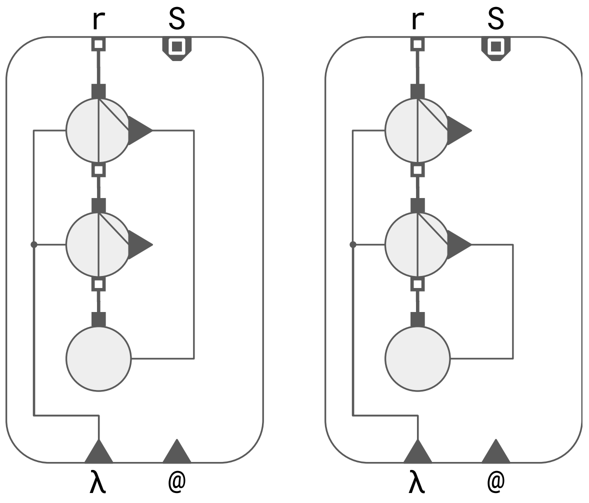
The type system also makes it easy to check a necessary well-formedness condition. In Figure [4], consider the gate produced by `b` on the right side. The wire connecting the "parent" port of the gate to the "right child" port makes this ideogram invalid as a binary tree. It is also not possible to write this in the pseudocode, as it would require feeding the output of a call to *b* back as the second argument to that same call. The graph structure on types lets us forbid this: informally, in order for an ideogram to be well-formed, there must be no cycles alternating between type edges and wires.[6]

[6]Formally, the condition is somewhat more relaxed: the type edges and wires must form a chorded R&B-cograph, as in [19]. That is, for every cycle there must be an edge not part of the cycle that connects two nodes in the cycle.

2.5 Representing Syntax Trees with Binding

We can finally provide a “Church encoding” of syntax trees with binding. For this example, we will consider closed terms in untyped lambda calculus. These trees have three kinds of nodes: application (`@`) with two children, abstraction (`λ`) with one child, and variable with no children. However, somehow each variable node must correspond to some abstraction node above it.

Analogous to how binary trees had factories for branch and leaf gates, here we have factories for application and abstraction gates. However, we do *not* start with a factory for variable gates; rather each abstraction gate will itself provide a factory which can be used to produce variable gates. The total number of factories will increase every time the abstraction factory produces a new gate. Finally, variables in lambda calculus can only be used in the body of their binding abstraction, not in some other subtree. By choosing the right edge relation in the type, the acyclicity condition of Section 2.4 will rule out this kind of malformed lambda calculus term.



$$\underbrace{\{\forall S\}}_{\mathtt{S}} \bowtie \underbrace{\{¡(\{\overline{S}\} \bowtie \{\overline{S}\} \bowtie \{\underline{S}\})\}}_{\mathtt{@}} \bowtie \underbrace{\{¡((\{\overline{S}\} \sqcup \{!\{\underline{S}\}\}) \bowtie \{\underline{S}\})\}}_{\lambda} \bowtie \underbrace{\{\underline{S}\}}_{\mathtt{r}}$$

Figure 5: The type of “Church encodings” of closed terms in untyped lambda calculus (bottom), and two inhabitants, $\lambda x.\lambda y.x$ (top left) and $\lambda x.\lambda y.y$ (top right). The lines on top of the gates are not part of the term, but are shown to reflect edges in the type of `λ`.

The type for closed terms in untyped lambda calculus and two example ideograms are shown in Figure 5. In both the type and the diagrams, `S` and `r` are the same as they were for binary trees, and `@` is the same as `b` was for binary trees. The substance is in the type of `λ`: $(\{\overline{S}\} \sqcup \{!\{\underline{S}\}\}) \bowtie \{\underline{S}\}$

The $!\{\underline{S}\}$ in the type gives each `λ` gate a factory port, which can itself produce “variable” gates. The “$(\cdot \sqcup \cdot) \bowtie \cdot$” produces the edge relation depicted on the `λ` gates in Figure 5, permitting wires between the $\overline{S}$ (hollow square) and $!\{\underline{S}\}$ (solid triangle) ports but forbidding those ports being wired to the $\underline{S}$ (solid square) port. This means that gates produced by the $!\{\underline{S}\}$ factory can only be used in the subtree beneath—ensuring that variable occurrences are well-scoped.

As analogues of Church encodings, these closed terms in untyped lambda calculus are functions that take one case per constructor (`@` and λ). The λ case must accept a value $\overline{S}$ from its body, provide a value $\underline{S}$ to its parent, and, most interestingly, also provide a factory $!\{\underline{S}\}$ which will be used once per variable occurrence. This allows for computation while respecting the binding structure, including providing capture-avoiding substitution for free.

3 Related Work

Graph Representations of Programming Languages There is work representing existing programming languages, in particular lambda calculus, as graphs [6, 20, 8]. 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 [1, 19]. Our type system is closely related to theirs when we do not use exponentials or second-order propositional quantifiers. (Though we adopt the edge convention opposite of theirs for $\otimes$ and $⅋$.)

Proof Nets and Interaction Nets Our work is closely related to proof nets [7], and in particular their extension, interaction nets [9]. There are two key differences between our work and interaction nets: (1) In interaction nets, each symbol (roughly our gates) has a *principal* port, which is used in reduction; in our work, gates 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 [13, 12, 15]. This work uses explicit "duplication" and "erasure" symbols, whereas exponentials (factories) are a central piece of our formalism. Moreover, these encodings require the interaction nets to have an infinite schema of symbols, whereas our richer type system can express the type of closed lambda terms concisely by using exponentials. A key advantage of that work is improved reduction performance on some benchmarks, facilitated by the sharing of subterms [12]. We hope to evaluate our system on their benchmark in future work.

There are versions of both proof nets [7] and interaction nets [10, 11] that represent exponentials with "boxes", and we expect these to be closely related to Ideograph. One clear difference is that the propositions in these systems are formulas in linear logic, whereas the types in our work are graphs. If Conjectures 1 and 2 (that our types quotient out certain type isomorphisms in polymorphic multiplicative exponential linear logic) hold, then we further conjecture that our terms are a quotienting of proof nets (e.g. removing $\otimes$ and $⅋$ nodes).

Representations of Graphs There is work representing graph structures in pure functional programming languages via recursive binders [14]. 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 [18], GP-2 [17], and LMNtal [21]. 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 [22], which extends LMNtal with hyperedges, has been used to encode lambda calculus terms. In contrast to our use of factories, they connect a binder to all of its variable occurrences via a single hyperedge.

4 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 like a system where types are canonical in a meaningful sense, and we believe our system may quotient out certain type isomorphisms of linear logic (Conjectures 1 and 2). We would also like to characterize the structures that can be defined in our types. Finally, we plan to develop an implementation, which we expect to be fairly straightforward, as our operational semantics consists solely of substituting nodes and edges and does not require difficult subgraph matching in the way that many graph rewriting systems do.

References

[1] 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. Available at `https://doi.org/10.1145/3373718.3394763`.

[2] 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: *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pp. 50–65.

[3] Nicolaas Govert de Bruijn (1972): *Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*. *Indagationes Mathematicae* 34, pp. 381–392.

[4] Roberto Di Cosmo (1991): *Invertibility of terms and valid isomorphisms. A proof theoretic study on second order λ-calculus with surjective pairing and terminal object*. Technical Report 91-10, LIENS - Ecole Normale Supérieure. Available at `http://www.dicosmo.org/TR/LIENS-93-11.pdf`.

[5] 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:https://doi.org/10.1016/j.entcs.2018.10.002. Available at `https://www.sciencedirect.com/science/article/pii/S1571066118300689`. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[6] Dan R. Ghica, Koko Muroya & Todd Waugh Ambridge (2019): *A robust graph-based approach to observational equivalence*, doi:10.48550/ARXIV.1907.01257. Available at `https://arxiv.org/abs/1907.01257`.

[7] Jean-Yves Girard (1987): *Linear logic*. *Theoretical Computer Science* 50(1), pp. 1–101, doi:https://doi.org/10.1016/0304-3975(87)90045-4. Available at `https://www.sciencedirect.com/science/article/pii/0304397587900454`.

[8] 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. Available at `https://doi.org/10.4204/EPTCS.288.1`.

[9] 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. Available at `https://doi.org/10.1145/96709.96718`.

[10] Yves Lafont (1995): *From Proof-Nets to Interaction Nets.* In: *Proceedings of the Workshop on Advances in Linear Logic*, Cambridge University Press, USA, p. 225–247.

[11] Ian Mackie (2000): *Interaction nets for linear logic.* *Theoretical Computer Science* 247(1), pp. 83–140, doi:https://doi.org/10.1016/S0304-3975(00)00198-5. Available at `https://www.sciencedirect.com/science/article/pii/S0304397500001985`.

[12] 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.

[13] Ian Craig Mackie (1994): *The Geometry of Implementation.* PhD thesis, Imperial College of Science, Technology and Medicine. Available at `https://spiral.imperial.ac.uk/handle/10044/1/46072`.

[14] 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. Available at `https://doi.org/10.1145/2364527.2364541`.

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

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

[17] 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. Available at `https://doi.org/10.4204/EPTCS.82.1`.

[18] 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.

[19] Christian Retoré (2003): *Handsome proof-nets: perfect matchings and cographs.* *Theoretical Computer Science* 294(3), pp. 473–488, doi:https://doi.org/10.1016/S0304-3975(01)00175-X. Available at `https://www.sciencedirect.com/science/article/pii/S030439750100175X`. Linear Logic.

[20] Ralf Schweimeier & Alan Jeffrey (1999): *A Categorical and Graphical Treatment of Closure Conversion.* *Electronic Notes in Theoretical Computer Science* 20, pp. 481–511, doi:https://doi.org/10.1016/S1571-0661(04)80090-2. Available at `https://www.sciencedirect.com/science/article/pii/S1571066104800902`. MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference.

[21] Kazunori Ueda (2009): *LMNtal as a hierarchical logic programming language.* *Theoretical Computer Science* 410(46), pp. 4784–4800, doi:https://doi.org/10.1016/j.tcs.2009.07.043. Available at `https://www.sciencedirect.com/science/article/pii/S0304397509005325`. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.

[22] 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.