# How to derive a type system from a term calculus and a collection

Michael Stay
Pyrofex Corp.
stay@pyrofex.net

L.G. Meredith
RChain Cooperative
greg@rchain.coop

June 22, 2017

**Abstract**

We present an algorithm for generating a powerful type system automatically, given a term calculus and a collection monad. The type system is sound and complete by construction. The Curry-Howard-Lambek isomorphism says that types correspond to predicates; in particular, types represent the collection of terms that satisfy the predicate. We can write down grammars for both term calculi and collections. By combining the two grammars we get a language for describing collections of terms. We can then add other features like modalities and recursion. Viewed as a logic, it is a language for expressing formulae; viewed as a type system, it is a Caires style spatial-behavioral type system. Different collections give rise to different logics. Lambek's contribution to the isomorphism was to show that types and and lambda terms modulo rewrites give cartesian closed categories. Our generated type system gives rise to a category with formulae as objects and witnesses of typing judgements as morphisms; it comes equipped with dinatural transformations for each inference rule, which in turn are generated by the rewrites in the calculus. We show that we can recover Lambek's denotational semantics of $\lambda-$calculus as a special case, and also that arrow types arise as a special case of a generalized possibility modality.

## 1 Introduction

Many widely-used programming languages in production today have weak static typing or none at all. Whatever its merits, dynamic typing makes it hard to detect type errors before the users of the program do. Microsoft's TypeScript [**?**], Facebook's Flow [**?**], and Google's Closure Compiler [**?**] are all massive projects trying to retrofit static types onto JavaScript; each company has found that static types are necessary for reliability in production. Similarly, the MyPy project [**?**] is trying to retrofit static types to Python.

Every type system of which we're aware was created by hand, and type systems are not always easy to define. Milner [**?**] was only able to give a simple sorting to the $\pi-$calculus. It took fourteen years to advance the field to the point that Caires was able to present a spatial-behavioral type system powerful enough to reason about the behavior of $\pi-$calculus terms rather than just the type of data sent on the channels [**?**].

We give an algorithm for generating a spatial-behavioral type system automatically from a presentation of the operational semantics of a term calculus and a grammar for a collection monad. The type system is sound and complete by construction.

The Curry-Howard-Lambek isomorphism famously says that types correspond to predicates. In 1920, Moses Schönfinkel was trying to reason about bound variables in logical statements with quantifiers, and came up with a clever idea: he could hide all uses of binders inside a small set of "combinators" and eliminate the need for the bound variables entirely.

$$\begin{aligned} S &= \lambda xyz.((x\ z)\ (y\ z)) \\ K &= \lambda yz.y \\ I &= \lambda z.z \end{aligned}$$

We can think of $S, K$, and $I$ as being "directions" to the location of a bound variable $z$. $S$ says that it may appear in either term of an application; $K$ says that it does not appear in a term; and $I$ says that it appears "right here".

In 1938, Haskell Curry noticed that the types of the combinators corresponded to axiom schemata in the Hilbert system for positive implicational logic:

$$\begin{aligned} S\colon &\quad \forall ABC.(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C \\ K\colon &\quad \forall AB.A \Rightarrow B \Rightarrow A \\ I\colon &\quad \forall A.A \end{aligned}$$

In 1969, William Howard noticed that intuitionistic natural deduction can be interpreted as a typed variant of $\lambda-$calculus. Shortly after, Lambek showed that types and equivalence classes of lambda terms with one free variable form a cartesian closed category and that every cartesian closed category can be presented this way.

The isomorphism works for predicates on any set, but when we consider predicates on the set of terms itself, something magical happens: we can think of types as being inhabited by the collection of terms that satisfy the predicate. Since we usually only care about those sets we can construct, we don't lose anything by insisting that predicates talk about terms.

We can write down grammars for both term calculi and collections. By combining the two grammars we get a language for describing collections of terms. We can then add other features like modalities and recursion. Viewed as a logic, it is a language for expressing formulae; viewed as a type system, it is a Caires style spatial-behavioral type system. Different collections give rise to different logics. Lambek's contribution to the isomorphism was to show that types and and lambda terms modulo rewrites give cartesian closed categories. Our generated type system gives rise to a category with formulae as objects and witnesses of typing judgements as morphisms; it comes equipped with dinatural transformations for each inference rule, which in turn are generated by the rewrites in the calculus. We show that we can recover Lambek's denotational semantics of $\lambda-$calculus as a special case.

# 2 blah

Denotational semantics works really well for functional programming languages. The actual process of computation is largely ignored in denotational semantics;

it doesn't matter *how* you compute the function, just *what* the function is. Lambek and Scott [**?**] constructed a cartesian closed category of types and $\alpha$-$\beta$-$\eta$-equivalence classes of terms with one free variable, and then assigned meaning to the types and terms with a cartesian closed functor into Set.

Denotational semantics gets a lot harder once we move away from functional programming languages. Modern programs run on multiple computers at the same time and each computer has several cores. The computers are connected by networks that can mix up the order in which messages are received. A program may run perfectly by itself, but deadlock when you run it in parallel with another copy of itself. The notion of "composition" begins to change, too: we run programs in parallel with each other and let them interact by passing messages back and forth, not simply by feeding the output of one function into the input of another. All of this makes it hard to think of such programs as functions.

Operational semantics is the other end of the spectrum, concerned with the rules by which the state of a computer changes. Whereas denotational semantics is inspired by Church and the $\lambda-$calculus, operational semantics is inspired by Turing and his machines. All of computational complexity lives here (e.g. $P \stackrel{?}{=} NP$).

To talk about the operational semantics of a programming language, there are five things we need to define.

First, we have to describe the layout of the state of the computer. For each kind of data that goes into a description of the state, we have a "sort". If we're using a programming language like $\lambda-$calculus, we have a sort for variables and a sort for terms, and the term is the entire state of the computer. If we're using a Turing machine, there are more parts: the tape, the state transition table, the current state, and the position of the read/write head on the tape. If we're using a modern language like JavaScript, the state is very complex: there are a couple of stacks, the heap, the lexical environment, the ¡code¿this¡/code¿ binding, and more.

Second, we have to build up the state itself using "term constructors". For example, in $\lambda-$calculus, we start with variables and use abstraction and application to build up a specific term.

Third, we say what rearrangements of the state we're going to ignore; this is called "structural congruence". In $\lambda-$calculus, we say that two terms are the same if they only differ in the choice of bound variables. In pi calculus, it doesn't matter in what order we list the processes that are all running at the same time.

Fourth, we give "reduction rules" describing how the state is allowed to change. In $\lambda-$calculus, the state only changes via $\beta$-reduction, substituting the argument of a function for the bound variable. In a Turing machine, each state leads to one of five others (change the bit to 0 or 1, then move left or right; or halt). In pi calculus, there may be more than one transition possible out of a particular state: if a process is listening on a channel and there are two messages, then either message may be processed first. Computational complexity theory is all about how many steps it takes to compute a result, so we do not have equations between sequences of rewrites.

Finally, the reduction rules themselves may only apply in certain "contexts"; for example, in all modern programming languages based on the $\lambda-$calculus, no reductions happen under an abstraction. That is, even if a term $t$ reduces to

$t'$, it is never the case that $\lambda x.t$ reduces to $\lambda x.t'$. The resulting normal form is called "weak head normal form" (WHNF).

Here's an example from Boudol's paper "[The $\pi$-calculus in direct style](http://www-sop.inria.fr/meije/personnel/Gerard.Boudol/popl97-abstract.html)." There are two sorts: $x$ or $z$ for variables and $L$ or $N$ for terms. The first line, labeled "syntax," defines four term constructors. There are equations for structural congruence, and there are two reduction rules followed by the contexts in which the rules apply:

We'd like to formalize this using category theory. For our first attempt, we capture almost all of this information in a multisorted Gph-enriched Lawvere theory: we have a generating object for each sort, a generating morphism for each term constructor, an equation between morphisms encoding structural congruence, and an edge for each rewrite.

We interpret the theory in Gph. Sorts map to graphs, term constructors to graph homomorphisms, equations to equations, and rewrites map to things I call "graph transformations", which are the obvious generalization of a natural transformation to the category of graphs: a graph transformation between two graph homomorphisms $\alpha : F \Rightarrow G$ assigns to each vertex $v$ an edge $\alpha_v : Fv \to Gv$. There's nothing about a commuting square in the definition because it doesn't even parse: we can't compose edges to get a new edge.

This initial approach doesn't *quite* work because of the way reduction contexts are usually presented. The reduction rules assume that we have a "global view" of the term being reduced, but the category theory insists on a "local view". By "local" I mean that we can always whisker a reduction with a term constructor: if $K$ is an endomorphism on a graph, then given any edge $e : v \to v'$, there's necessarily an edge $Ke : Kv \to Kv'$. These two requirements conflict: to model reduction to WHNF, if we have a reduction $t \to t'$, we don't want a reduction $\lambda x.t \to \lambda x.t'$.

One solution is to introduce "context constructors", unary morphisms for marking reduction contexts. These contexts become part of the rewrite rules and the structural congruence; for example, taking $C$ to be the context constructor for WHNF, we add a structural congruence rule that says that to reduce an application of one term to another, we have to reduce the term on the left first:

$$C(T\ U) \equiv (CT\ U).$$

We also modify the reduction reduction rule to involve the context constructors. Here's $\beta$ reduction when reducing to WHNF:

$$\beta : (C(\lambda x.T)\ U) \Rightarrow CT\{U/x\}.$$

Now $\beta$ reduction can't happen just anywhere; it can only happen in the presence of the "catalyst" $C$.

With context constructors, we *can* capture all of the information about operational semantics using a multisorted Gph-enriched Lawvere theory: we have a generating object for each sort, a generating morphism for each term constructor and for each context constructor, equations between morphisms encoding structural congruence and context propagation, and an edge for each rewrite in its appropriate context.

We can recover the Lambek/Scott-style denotational semantics of the SK calculus (see the appendix) by taking the Gph-theory, modding out by the

edges, and taking the monoid of endomorphisms on the generating object. The monoid is the cartesian closed category with only the "untyped" type. Using [Mellis and Zeilberger's notion of a functor as a type refinement system](http://www.irif.fr/ mellies/papers/functors-are-type-refinement-systems.pdf), we "-oidify" the monoid into a category of types and equivalence classes of terms.

However, modding out by edges utterly destroys the semantics of concurrent languages, and composition of endomorphisms doesn't line up particularly well with composition of processes, so neither of those operations are desirable in general. That doesn't stop us from considering Gph-enriched functors as type refinement systems, though.

Let $G$ be the free model of a theory on the empty graph. Our plan for future work is to show how different notions of a collection of edges of $G$ give rise to different kinds of logics. For example, if we take subsets of the edges of $G$, we get subgraphs of $G$, which form a Heyting algebra. On the other hand, if we consider sets of lists of composable edges in $G$, we get quantale semantics for linear logic. Specific collections will be the types in the type system, and proofs should be graph homomorphisms mapped over the collection. Edges will feature in proof normalization.

At the end, we should have a system where given a [formal semantics for a language](http://www.kframework.org/index.php/Projects), we algorithmically derive a type system tailored to the language. We should also get a nice Curry-Howard style approach to operational semantics that even denotational semantics people won't turn up their noses at!

For our purposes, a **graph** is a set $E$ of edges and a set $V$ of vertices together with three functions $s\colon E \to V$ for the source of the edge, $t\colon E \to V$ for the target, and $a\colon V \to E$ such that $s \circ a = V$ and $t \circ a = V$—that is, $a$ assigns a chosen self-loop to each vertex. A **graph homomorphism** maps vertices to vertices and edges to edges such that the source and target are preserved. **Gph** is the category of graphs and graph homomoprhisms. Gph has finite products: the terminal graph is the graph with one vertex and one loop, while the product of two graphs $(E, V, s, t, a) \times (E', V', s', t', a')$ is $(E \times E', V \times V', s \times s', t \times t', a \times a')$.

Gph is a topos; the subobject classifier has two vertices $t, f$ and five edges: the two self-loops, an edge from $t$ to $f$, an edge from $f$ to $t$, and an extra self-loop on $t$. Any edge in a subgraph maps to the chosen self-loop on $t$, while an edge not in the subgraph maps to one of the other four edges depending on whether the source and target vertex are included or not.

A **Gph-enriched category** consists of

- a set of objects; - for each pair of objects $x, y$, a graph $\hom(x, y)$; - for each triple of objects $x, y, z$, a composition graph homomorphism $\circ\colon \hom(y, z) \times \hom(x, y) \to \hom(x, z)$; and - for each object $x$, a vertex of $\hom(x, x)$, the identity on $x$,

such that composition is associative, and composition and the identity obey the unit laws. A Gph-enriched category has finite products if the underlying category does.

Any category is trivially Gph-enrichable by treating the elements of the hom sets as vertices and adjoining a self loop to each vertex. The category Gph is nontrivially Gph-enriched: Gph is a topos, and therefore cartesian closed, and therefore enriched over itself. Given two graph homomorphisms $F, F'\colon (E, V, s, t, a) \to (E', V', s', t', a')$, a **graph transformation** assigns to each vertex $v$ in $V$ an edge $e'$ in $E'$ such that $s'(e') = F(v)$ and $t'(e') = F'(v)$.

Given any two graphs $G$ and $G'$, there is an **exponential graph** $G'^G$ whose vertices are graph homomorphisms between them and whose edges are graph transformations. There is a natural isomorphism between the graphs $C^{A \times B}$ and $(C^B)^A$.

A **Gph-enriched functor** between two Gph-enriched categories $C, D$ is a functor between the underlying categories such that the graph structure on each hom set is preserved, *i.e.* the functions between hom sets are graph homomorphisms between the hom graphs.

Let $S$ be a finite set, FinSet be a skeleton of the category of finite sets and functions between them, and FinSet$/S$ be the category of functions into $S$ and commuting triangles. A **multisorted Gph-enriched Lawvere theory**, hereafter **Gph-theory** is a Gph-enriched category with finite products Th equipped with a finite set $S$ of **sorts** and a Gph-enriched functor $\theta \colon \mathrm{FinSet}^{\mathrm{op}}/S \to \mathrm{Th}$ that preserves products strictly. Any Gph-theory has an underlying multisorted Lawvere theory given by forgetting the edges of each hom graph.

A **model** of a Gph-theory Th is a Gph-enriched functor from Th to Gph that preserves products up to natural isomorphism. A **homomorphism of models** is a braided Gph-enriched natural transformation between the functors. Let FPGphCat be the 2-category of small Gph-enriched categories with finite products, product-preserving Gph-functors, and braided Gph-natural transformations. The forgetful functor $U \colon \mathrm{FPGphCat}[\mathrm{Th}, \mathrm{Gph}] \to \mathrm{Gph}$ that picks out the underlying graph of a model has a left adjoint that picks out the free model on a graph.

Gph-enriched categories are part of a spectrum of 2-category-like structures. A strict 2-category is a category enriched over Cat with its usual product. Sesquicategories are categories enriched over Cat with the "funny" tensor product; a sesquicategory can be thought of as a generalized 2-category where the interchange law does not always hold. A Gph-enriched category can be thought of as a generalized sesquicategory where 2-morphisms (now edges) cannot always be composed. Any strict 2-category has an underlying sesquicategory, and any sesquicategory has an underlying Gph-enriched category; these forgetful functors have left adjoints.

Here's a presentation of the Gph-theory for the SK calculus:

- objects - $T$ - morphisms - $S \colon 1 \to T$ - $K \colon 1 \to T$ - $(- \ -) \colon T \times T \to T$ - equations - none - edges - $\sigma \colon (((S\ x)\ y)\ z) \Rightarrow ((x\ z)\ (y\ z))$ - $\kappa \colon ((K\ x)\ z) \Rightarrow x$

The free model of this theory on the empty graph has a vertex for every term in the SK calculus and an edge for every reduction.

Here's the theory above modified to account for WHNF.

- objects - $T$ - morphisms - $S \colon 1 \to T$ - $K \colon 1 \to T$ - $(- \ -) \colon T \times T \to T$ - $R \colon T \to T$ - equations - $R(x\ y) = (Rx\ y)$ - edges - $\sigma \colon (((RS\ x)\ y)\ z) \Rightarrow ((Rx\ z)\ (y\ z))$ - $\kappa \colon ((RK\ x)\ z) \Rightarrow Rx$

If $M$ is an $SK$ term with no uses of $R$ and $M'$ is its weak head normal form, then $RM$ reduces to $RM'$.

Once we have $R$, we can think about other things to do with it. The rewrites treat the morphism $R$ as a linear resource. If we treat $R$ as "fuel" rather than "catalyst"—that is, if we modify $\sigma$ and $\kappa$ so that $R$ only appears on the left-hand side—then the reduction of $R^n M$ counts how many steps the computation takes; this is similar to Ethereum's notion of "[gasoline](https://www.cryptocompare.com/coins/guides/what-is-the-gas-in-ethereum/)."

# 3 Appendix

Here we review some standard definitions and results in enriched category theory; see [**?**], [**?**], [**?**], and [**?**] for more details.

A **directed multigraph with self loops**, hereafter **graph**, consists of a set $E$ of edges, a set $V$ of vertices, two functions $s, t\colon E \to V$ picking out the source and target of each edge, and a function $a\colon V \to E$ such that $s \circ a$ and $t \circ a$ are both the identity on $V$—that is, $a$ equips each vertex in $V$ with a chosen self loop. There are no constraints on $E, V, s$, or $t$, so a graph may have infinitely many vertices and infinitely many edges between any pair of vertices. A **graph homomorphism** from $(E, V, s, t, a)$ to $(E', V', s', t', a')$ is a pair of functions $(\epsilon\colon E \to E', \upsilon\colon V \to V')$ such that $\upsilon \circ s = s' \circ \epsilon$ and $\upsilon \circ t = t' \circ \epsilon$. **Gph** is the category of graphs and graph homomorphisms. Gph has finite products: the terminal graph is the graph with one vertex and one loop, while the product of two graphs $(E, V, s, t, a) \times (E', V', s', t', a')$ is $(E \times E', V \times V', s \times s', t \times t', a \times a')$.

A **Gph-enriched category** consists of

- a set of objects;
- for each pair of objects $x, y$, a graph $\hom(x, y)$;
- for each triple of objects $x, y, z$, a composition graph homomorphism $\circ\colon \hom(y, z) \times \hom(x, y) \to \hom(x, z)$; and
- for each object $x$, a vertex of $\hom(x, x)$, the identity on $x$,

such that composition is associative, and composition and the identity obey the unit laws. A Gph-enriched category has finite products if the underlying category does.

Any category is trivially Gph-enrichable by treating the elements of the hom sets as vertices and adjoining a self loop to each vertex. The category Gph is nontrivially Gph-enriched: Gph is a topos, and therefore cartesian closed, and therefore enriched over itself. Given two graph homomorphisms $F, F'\colon (E, V, s, t, a) \to (E', V', s', t', a')$, a **graph transformation** assigns to each vertex $v$ in $V$ an edge $e'$ in $E'$ such that $s'(e') = F(v)$ and $t'(e') = F'(v)$. Given any two graphs $G$ and $G'$, there is an exponential graph $G'^G$ whose vertices are graph homomorphisms between them and whose edges are graph transformations.

A **Gph-enriched functor** between two Gph-enriched categories $C, D$ is a functor between the underlying categories such that the graph structure on each hom set is preserved, *i.e.* the functions between hom sets are graph homomorphisms between the hom graphs.

Let $S$ be a finite set, FinSet be a skeleton of the category of finite sets and functions between them, and FinSet$/S$ be the category of functions into $S$ and commuting triangles. A **multisorted Gph-enriched Lawvere theory**, hereafter **Gph-theory** is a Gph-enriched category with finite products Th equipped with a finite set $S$ of **sorts** and a Gph-enriched functor $\theta\colon \mathrm{FinSet}^{\mathrm{op}}/S \to \mathrm{Th}$ that preserves products strictly. Any Gph-theory has an underlying multisorted Lawvere theory given by forgetting the edges of each hom graph.

A **model** of a Gph-theory Th is a Gph-enriched functor from Th to Gph that preserves products up to natural isomorphism. A **homomorphism of models** is a braided Gph-enriched natural transformation between the functors.

Let FPGphCat be the 2-category of small Gph-enriched categories with finite products, product-preserving Gph-functors, and braided Gph-natural transformations. The forgetful functor $U : \mathrm{FPGphCat}[\mathrm{Th}, \mathrm{Gph}] \to \mathrm{Gph}$ that picks out the underlying graph of a model has a left adjoint that picks out the free model on a graph.

Gph-enriched categories are part of a spectrum of 2-category-like structures. A strict 2-category is a category enriched over Cat with its usual product. Sesquicategories are categories enriched over Cat with the "funny" tensor product [**?**]; a sesquicategory can be thought of as a 2-category where the interchange law does not hold. A Gph-enriched category can be thought of as a sesquicategory where 2-morphisms (now edges) cannot be composed. Any strict 2-category has an underlying sesquicategory, and any sesquicategory has an underlying Gph-enriched category; these forgetful functors have left adjoints.

# References