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 23, 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 of the calculus. By construction, we get a cut-elimination theorem. We show that we can recover Lambek's denotational semantics of λ-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 π-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 π-calculus terms rather than just the type of data sent on the channels [?].

In this paper, we will 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 resulting type system is sound and complete by construction.

Given a presentation of a term calculus, we can construct a graph G whose vertices are terms of the calculus and whose edges are rewrites. If we add "do-nothing" rewrites to each vertex, we can describe the whole graph just in terms of its edges. The Curry-Howard-Lambek isomorphism famously says that types correspond to predicates. The isomorphism works for predicates on any set, but when we consider predicates on the set of edges in G, something magical happens: we can think of types as being inhabited by the collection of edges that satisfy the predicate—in other words, types describe the behavior of programs.

Predicates need to talk about terms and rewrites as well as be able to assemble them together into a collection. Usually we take the collection to be sets of edges because we want to consider subgraphs of G, which form a Heyting algebra. Since we have a grammar for the edges of G and a grammar for Heyting algebras, we can consider the sum of the grammars to be a language for expressing predicates.

To this language we can add modalities. If ϕ is a predicate that picks out some subgraph of G, we can define $\diamond\phi$ to be a predicate satisfied by the set of terms that possibly reduce to a term in ϕ; similarly, we can define to be a predicate satisfied by the set of terms that necessarily reduce to a term in ϕ. We'll show later that a generalization of the "possibly" modality that is parametric in a two-hole term context gives rise both to the familiar arrow types of applicative calculi as well as Caires' rely-guarantee modality.

Denotational semantics asks what function a term denotes, and works really well for functional programming languages. The actual process of computation is largely ignored: it doesn't matter *how* you compute the function, just *what* the function is. 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 λ-calculus, operational semantics is inspired by Turing and his machines: every virtual machine is an executable spec for an operational semantics. Operational semantics has been criticized by advocates of denotational semantics as being completely ad-hoc and hard to reason about. In response, the operational semantics community developed tools like K framework [**?**] and Maude [**?**], principled approaches that make reasoning about operational semantics much easier.

Categorical semantics has traditionally been very function-oriented: Lambek's contribution [**?**] to the Curry-Howard-Lambek isomorphism was to show how to construct a cartesian closed category of types and equivalence classes of lambda terms with one free variable, and then assign denotations to the types and terms with a cartesian closed functor into Set. Despite this tradition, categorical semantics generalizes very easily. For example, when we generalize from the cartesian product to an arbitrary tensor product, we get symmetric monoidal closed categories, whose internal language is *linear* λ-calculus. Linear λ-calculus has models in the category Hilb of Hilbert spaces and linear transformations, so we get a notion of a quantum programming language.

Generalizing from categories to enriched categories lets us use Lambek's techniques to capture information about operational semantics; we enrich over graphs, so rather than forming a mere set of terms, we can form a graph of terms and rewrites. Our generated type system gives rise to a category with predicates as objects and witnesses of typing judgements as morphisms; the category comes equipped with dinatural transformations for each inference rule, which in turn are generated by the rewrites in the calculus. By construction, we get a cut-elimination theorem. Given such a categorical representation of the operational semantics of a functional programming language, we can throw away information about how the function is computed and, like Lambek, look for denotational semantics in Set; but given a categorical representation of the operational semantics of a nondeterministic, concurrent programming language like π-calculus, we can still look at functors to categories that preserve the relevant structure.

2 Operational semantics

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 λ-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 λ-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 λ-calculus, we say that two terms are the same if they only differ in the choice of bound variables. In π-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 λ-calculus, the state only changes via β-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 π-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 λ-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 π-calculus in direct style" [?]. 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:

$L ::= x \quad \mid \quad \lambda x L \quad \mid \quad (Lx) \quad \mid \quad (\mathrm{def}\, x = L \,\mathrm{in}\, L)$	syntax
$x \neq z \Rightarrow (\mathrm{def}\, x = N \,\mathrm{in}\, L)z \equiv (\mathrm{def}\, x = N \,\mathrm{in}\, Lz)$	structural congruence
$(\lambda x L)z \to [z/x]L$	reduction
$(\mathrm{def} \cdots x = L \cdots \mathrm{in}\, x y_1 \cdots y_n) \to (\mathrm{def} \cdots x = L \cdots \mathrm{in}\, L y_1 \cdots y_n)$	
$L \to L' \Rightarrow (Lz) \to (L'z)$	context rules
$L \to L' \Rightarrow (\mathrm{def}\, x = N \,\mathrm{in}\, L) \to (\mathrm{def}\, x = N \,\mathrm{in}\, L')$	
$L \to L' \& N \equiv L \Rightarrow N \to L'$	

3 Categorical semantics

In Lawvere's seminal 1963 thesis [?], he showed that categories with finite products suffice to model all of universal algebra. If a mathematical gadget can be described as sets equipped with functions satisfying axioms, there is a finite-products category Th such that all gadget arise as product-preserving functors from Th to Set, and each finite-products category defines a gadget. Monoids, groups, rings, natural numbers, and graphs are all examples; fields are a notable exception that cannot be defined this way. A "Gph-theory", defined below, is a Lawvere theory in which we can specify directed edges between morphisms without having to declare them equal; this lets us encode the notion of a reduction between terms. We can think of a presentation of a Gph-theory as a specification file for a toy version of K framework.

We should say something about binders. Reasoning about languages with binders is a hard-enough problem that it was the focus of the 2005 PoplMark challenge [?]. Recently, Pitts and Gabbay [?] developed the theory of nominal sets, and Clauston [?] defined nominal Lawvere theories. We anticipate that enriching nominal Lawvere theories in Gph will allow us to model the operational semantics of languages with binders, but in this paper, we restrict ourselves to languages without. This is not a severe restriction: we can eliminate abstractions from applicative calculi like λ-calculus by using Curry's S and K combinators; in a similar way we can eliminate input prefixes from concurrent calculi like π-calculus using Yoshida's combinators [?] and uses of the "new name" binder using the reflective techniques of Meredith and Radestock [?].

3.1 Gph-theories

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) Lawvere theory** is a category with finite products Th equipped with a finite set S of **sorts** and a functor $\theta\colon \mathrm{FinSet}^{\mathrm{op}}/S \to \mathrm{Th}$ that preserves products strictly.

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. 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 $\upsilon \circ s = s' \circ \epsilon$, $\upsilon \circ t = t' \circ \epsilon$, and $\epsilon \circ a = a' \circ \upsilon$. **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 collection 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 non-trivially Gph-enriched: 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.

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 monoidal 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 monoidal 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 hold. A Gph-enriched category can be thought of as a generalized 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.

3.1.1 Example: SK calculus

Here’s a presentation of the Gph-theory for the SK calculus with reduction to weak head normal form. We have modeled reduction contexts with a morphism R; given a term t with no occurrences of R, the term Rt reduces to Rt', where t' is the weak head normal form of t.

- objects
  - T, for terms
- morphisms
  - $S\colon 1 \to T$
  - $K\colon 1 \to T$
  - $(-\ -)\colon T \times T \to T$
  - $R\colon T \to T$, for reduction contexts
- equations
- $R(x\ y) = (Rx\ y)$, *i.e.* to reduce a term, you reduce the applicand but not the term it’s applied to
- edges
- $\sigma\colon (((RS\ x)\ y)\ z) \Rightarrow ((Rx\ z)\ (y\ z))$
- $\kappa\colon ((RK\ x)\ z) \Rightarrow Rx$

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

3.2 Lawvere theory of edges

Given a Gph-theory Th, define the Lawvere theory

- equations (Curry)
  - K = S(S(KS)(S(KK)K))(K(SKK))
  - S = S(S(KS)(S(K(S(KS)))(S(K(S(KK)))S)))(K(K(SKK)))
  - S(KK) = S(S(KS)(S(KK)(S(KS)K)))(KK)
  - S(KS)(S(KK)) = S(KK)(S(S(KS)(S(KK)(SKK)))(K(SKK)))
  - S(K(S(KS)))(S(KS)(S(KS))) = S(S(KS)(S(KK)(S(KS)(S(K(S(KS)))S))))(KS)

References