Categorical Semantics for Dynamically Typed Languages, Notes for History of Programming Languages, 2017

Max S. New, Northeastern University

April 30, 2017

Abstract

These are the notes for a talk I gave for Matthias Felleisen's *History of Programming Languages* class. I've tried to avoid recounting detailed descriptions of category theory and domain theory here, which I think the cited papers themselves did a good job of doing. Instead, I've tried to take advantage of the connections between syntax and category theory to reframe some of the results in these papers as syntactic translations, especially the theorems in 4, 7.

1 Historical Overview

In 1969, Dana Scott wrote a paper in which he said untyped lambda calculus had no mathematical meaning (Scott [1993]), 11 years later he wrote a paper that organized many of the different semantics he and others had since found using the language of category theory (Scott [1980]). This latter paper is really the first deserving of the title "categorical semantics of dynamic typing", and so I'm going to present some of the theorems and "theorems" presented in that paper, but mingled with the history of the idea and the preceding papers that led to them.

In Figure 1 we have a very skeletal timeline: On the left are some foundational categorical logic papers, on the right are some seminal semantics papers by Dana Scott that we'll go over in some detail throughout this article.

I drew them in parallel, but it is clear that there was interaction between the sides, for instance Scott says he received a suggestion by Lawvere in Scott [1972]. But what's very interesting is that *both* Lambek and Scott wrote papers on cartesian closed categories and lambda calculus for the 1980 Haskell Curry Festschrift (Lambek [1980] and Scott [1980]), so this seems to have been something of a milestone for the use of category theory in programming languages. For completeness, since I won't discuss them in detail below, the Lawvere paper is Lawvere [1969] where Lawvere introduced the notion of hyperdoctrine, which allows for the interpretation of higher-order logic. The Lambek papers form a

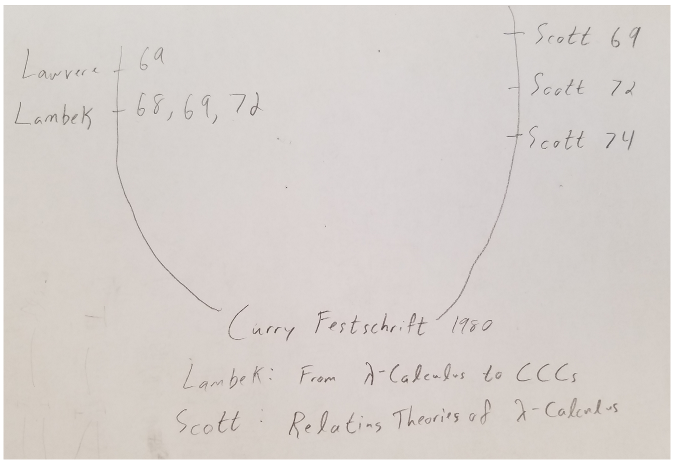

Figure 1: Artisinally Crafted Timeline

series called "Deductive Systems and Categories I,II,III" (Lambek [1968], Lambek [1969], Lambek [1972]), though I confess I wasn't able to find copies, so I'm not quite sure which one can really be said to be the "origin" of the theorem discussed in section 3.

2 Pre-history: Scott 1969

We start in 1969, with a really remarkable unpublished manuscript by Dana Scott "A type-theoretical alternative to ISWIM, CUCH, OWHY". This was a very influential paper, despite never being published, both Mitch Wand and Matthias Felleisen confirmed to me that they had read it. Basically, it lays out Dana Scott's philosophy of programming language semantics and already contains a fair amount of the foundations of domain theory.

Dana Scott opposed the "formalist" approach to combinatory logic and programming language semantics wherein you just define a syntax and declare some axioms that it should satisfy without some external justification. He includes in this camp combinator logic (SKI) and the untyped lambda calculus with β and η laws.

His main objection is that the axioms of said systems have no meaning independent of the syntax itself. He even says that we are lucky that these systems are consistent at all, and criticizes the Church-Rosser theorem, a syntactic method that proves that β, η equality is not trivial, as a brittle approach.

Instead he promotes a "model-theoretic" approach, in which you start with an intended mathematical semantics, and then justify your syntax as constructions in the model and justify your axioms by equations that hold in that model. In particular, the model should be "independent" from the syntax of the language itself so that the equalities in the models are just “ordinary mathematics”. In particular, it is desirable that the interpretation of application in the lambda calculus be “real” function application.

Most interesting to us is that he states that he believes *untyped* lambda calculus “makes no sense whatsoever”. This is why the paper was never published: he discovered denotational models of untyped lambda calculus later that very same year!

The paper was eventually published in *Theoretical Computer Science* with historical commentary from Scott. I highly recomend this version to anyone that works on programing language semantics.

3 Typed Lambda Calculi and Cartesian Closed Categories

So, just as Dana Scott did, let’s start with the models of *typed lambda calculus*, also the first theorem in (Scott [1980]).

You’ve probably heard this one before: they’re the cartesian closed categories. Now today typed lambda calculus can mean just about anything, so to be more precise, the theorem says that the call-by-name, negative, extensional typed lambda calculus corresponds exactly to cartesian closed categories.

By call-by-name I mean that we have β-reduction with arbitrary expressions being substituted $(\lambda x.t)u = t[u/x]$ for any expression u.

By negative, I mean that the types that correspond to structures present in *all* cartesian closed categories are the unit type 1, product type $A \times B$ and function type $A \to B$. And by extensional I mean that you have β and η equality for all these connectives.

For other flavors of lambda calculus (call-by-value, with sums and the empty type), you have a different theorem. But this one applied to the languages Dana Scott studied and was compatible with general recursion.

I don’t want to go into much detail because this is a very well-known result (see Lambek and Scott [1988]) Without going into too much detail (we are supposed to talk about dynamically typed languages after all), here’s a formulation of the correspondence:

Theorem 1 (Lambek)**.** *Constructing a cartesian closed category is the same as giving a model of the call-by-name, negative, extensional typed lambda calculus.*

This theorem is very nice for the semanticist because the definition of cartesian closed category is easier to use than all of the syntax of lambda calculus. The proof of this theorem is also not entirely trivial, which is why it is so useful, because contexts involve some amount of encoding to interpret using the cartesian structure of the category.

However in this talk, we’re going to use it in the opposite way. Since there’s a correspondence, any theorem about constructing a cartesian closed category can be rewritten as a translation from lambda calculus syntax to a model. If that model itself is syntactic, then we just have a syntactic translation!

4 Untyped Lambda Calculus is modelled by Unityped Lambda Calculus

Ok, time for our first real construction. How do we give a model of untyped lambda calculus? Well, remember we want to interpret the calculus' application as real function application, and every term in the lambda calculus can be applied, so everything in untyped lambda calculus must be a function. On the other hand everything can be applied to anything so if we were to give these terms a type it would have to be a type D so that at least we can interconvert between $D \to D$ and D. To be precise we will assume we are talking about some CCC with an object D we will call a *reflexive object*, that has two functions

$$e : (D \to D) \to D \qquad p : D \to (D \to D)$$

What relationship do these need to have in order to model untyped lambda calculus? Isomorphism, something weaker? Let's find out, with the goal of making the following theorem true:

Theorem 2 (Scott)**.** *A Cartesian Closed Category with a Reflexive Object is a model of Untyped Lambda Calculus.*

By untyped lambda calculus I mean pure lambda calculus, just application and lambda.

Since we are giving a model of untyped lambda calculus, that means to prove the theorem we need to translate untyped lambda calculus into typed lambda calculus with a reflexive object. So we will consider e, p to be constants of the types given above and we will figure out what additional equations we need.

Let's define the translation. The translation function will be called $C(-)$. We will compile an untyped lambda term $x_1, \dots, x_n \vdash t$ to a typed lambda term $x_1 : D, \dots, x_n : D \vdash C(t) : D$.

$$\begin{array}{rcl} C(x) & = & x \\ C(\lambda x.t) & = & e(\lambda x : D.C(t)) \\ C(tu) & = & (p(C(t)))C(u) \end{array}$$

To show this is a model of untyped lambda calculus, we need to show that we preserve equivalence of untyped lambda terms. There are of course multiple choices here. The original choice is to have β and η, aka "the" lambda calculus, which I will call the "untyped lambda calculus". However, this is not so relevant to real dynamically typed programming languages, which have many different data types, these would have just β equality for functions and η will fail for those terms that are not functions.

Let's start by preserving β. We need to show that

$$C((\lambda x.t)u) = C(t[u/x])$$

So let's start with the left side:

$$C((\lambda x.t)u) \quad = \quad (p(e(\lambda x : D.C(t))))C(u)$$

We'd really like to get that $C(u)$ to be substituted into that lambda, so to do that we will need to add an axiom to D: that $p(e(x)) \equiv x$. The category-theoretic terminology is to say that p is a *retraction* of e, or that e is a *section* of p. Collectively, we'll call them a *retract*. If e, p form a retract, then we get

$$\begin{array}{rcl} C((\lambda x.t)u) & = & (p(e(\lambda x : D.C(t))))C(u) \\ & = & (C(t))[C(u)/x] \\ & = & C(t[u/x]) \end{array}$$

Where the last step follows from an easy compositionality theorem for our translation.

What about η, that is $t = \lambda x.(tx)$? Let's calculate again:

$$\begin{array}{rcl} C(\lambda x.(tx)) & = & e(\lambda x : D.p(C(t))x) \\ & = & e(p(C(t))) \end{array}$$

where the second step is the *typed* η principle (remember $p(C(t)) : D \to D$). Then clearly we need $e(p(x)) = x$. Together with the retraction property, this would make e, p into an *isomorphism*.

We'll call an object D such that $D \to D$ is isomorphic to D an *extensional* reflexive object, and when e, p is just a section-retraction pair, we'll call it an *intensional* reflexive object. Then we can refine our theorem before to pick which equations we want:

Theorem 3. 1. *A CCC with an* extensional *reflexive object is a model of untyped lambda calculus with β and η equality.*

2. *A CCC with an* intensional *reflexive object is a model of untyped lambda calculus with β equality.*

5 Constructing a Reflexive Object: D_∞

Ok, so that explains how we can get untyped models from typed models with a reflexive object, but that doesn't tell us that there are any "mathematical" models like Scott desired.

There's a lesson here: at its most abstract, category theory is very "syntactic". In fact in several of the earlier papers, Dana Scott expresses a mild disdain for category theory, but by 1980 he had certainly accepted it as a crucial tool for the semanticist.

So let's get a high level view of some concrete[1] models of untyped lambda calculus, the very first ones, from Scott [1972].

[1]concrete has two completely opposite meanings in programming languages, to some syntax is concrete, to others semantics

At first, we seem pretty doomed by Cantor's diagonalization argument, how can we possibly get a set where $D \cong D \to D$? Just think about the cardinalities, we would have $|D| = |D|^{|D|}$, so the only solutions where D is a set and $D \to D$ is the set of all functions from D to D are where $D = \emptyset, \{*\}$, so not useful as a model of lambda calculus, which we know can encode all natural numbers and partial computable functions on them.

But, remember $D \to D$ is an object axiomatized by the notion of a cartesian closed category, there's nothing that says it has to mean the set of *all* functions from a set D to itself, and Dana Scott evades the paradox by considering only the *continuous* functions.

Now, I won't go into the full details, but suffice it to say that Scott defined something called *continuous lattices*, which are sets with extra order-theoretic/topological structure that includes an element $\bot$ that denotes "undefined" or "divergence". Next he defined continuous functions on continuous lattices that basically ensures that if you want finite information about the output of a continuous function, you only need to provide finite information about the input, which is clearly a criterion that all *computable* functions meet.

Then we can ask how do we get a D where $D \cong D \to D$, where $\to$ now means *continuous* functions? Well if that holds, then

$$D \cong (D \to D) \cong ((D \to D) \to (D \to D)) \cong \cdots$$

So the idea is to build up something that looks like an infinite tree of $\to$s, an *fixed point* of the $\to$ "function". Then we can hope to build a fixed point a la Tarski by starting with a continuous lattice D_0 and iterating:

$$\begin{aligned}
&D_0\\
&D_1 = (D_0 \to D_0)\\
&D_2 = (D_1 \to D_1)\\
&\quad\vdots
\end{aligned}$$

In Tarski's theorem we would have $D_i \leq D_{i+1}$, and the equivalent we need to use is what's called an embedding-projection pair, which is a refinement of the idea of a section-retraction pair. An embedding-projection pair from A to B is a pair of $e : A \to B$ (the embedding), and $p : B \to A$ (the projection) such that $p \circ e = \mathrm{id}_A$ (so they form a section-retraction pair), and $e \circ p \leq \mathrm{id}_B$. We will refer to an embedding-projection pair collectively as an e-p pair and denote an e-p pair from A to B as $(e, p) : A \lhd B$.

So to construct a D_∞, we start with any D_0 with a projection $(e_0, p_0) : D_0 \lhd (D_0 \to D_0)$ (for example the ordered truth values work), and we start iterating, by defining:

$$e_{i+1} : (D_{i+1} = (D_i \to D_i)) \to (D_{i+1} \to D_{i+1})$$

$$e_{i+1}(x)(y) = e_i(x(p_i(y)))$$

$$p_{i+1} : (D_{i+1} \to D_{i+1}) \to (D_{i+1} = (D_i \to D_i))$$

$$p_{i+1}(x)(y) = p_i(y(e_i(x)))$$

And then we take D_∞[2] to be the "inverse limit" of the projections p. So what does an element of D_∞ look like? Well it's just an infinite vector of elements from each D_i such that they agree when you project:

$$D_\infty = \{(x_i)_{i \in \mathbb{N}} | p_i(x_{i+1}) = x_i\}$$

which is quite a nasty looking space, but does the job of satisfying $D_\infty = D_\infty \to D_\infty$.

We can apply the same inverse limit construction to get models of dynamically typed languages with some booleans by instead constructing roughly $D'_\infty = \mathbb{B} \oplus (D'_\infty \to D'_\infty)$ and research on solving these "recursive domain equations" went on for quite some time (Wand [1979], Smyth and Plotkin [1982],Pitts [1996] to name a few), and has even seeped into the operational world in the form of step-indexed logical relations (Appel and McAllester [2001], Ahmed [2004]).

So in summary, the D_∞ construction is nice because it obviously gives a model, and we can construct many other similar models using the same idea.

On the other hand, the elements we get are complicated, and the abundance of different constructions we got are not obviously comparable. How do I know that I got *all* of the functions I want in my model?

6 Constructing a Universal Object: P_ω

So using the D_∞ construction we can build many different models of untyped lambda, but which ones do we actually want? If we want to pick a single untyped universe of values, we'd like it to support as many constructions as possible.

In other words, we'd like some idea of what *types* our untyped model can represent, and even better we'd like to get untyped models that represent exactly some natural classes of domains that we already have.

To answer this question, we need to understand 2 things. What does it mean to represent a type in another? And how could we possibly get one of these? These ideas are all in Scott [1976].

[2]pronounced "dee infinity"

6.1 Types as Retracts as Idempotents

When can one type represent another? Let's say I have a dynamic type D, and I'd like to be able to represent all of the operations of some types $X, Y, Z, \ldots$. So for example every function $X \times Y \to Z$ should be implementable as a function $D \times D \to D$, *without* exposing intensional details about how the function is implemented.

Retracts work very well for this purpose. To see why, let's say we have retractions for X and Y into D, that is we have section retraction pairs $e_X : X \to D, p_X : D \to X$ with $p_X \circ e_X = \mathrm{id}_X$ and similarly e_Y, p_Y. Then every function $f : X \to Y$ can be encoded as $p_Y \circ f \circ e_X$. Furthermore, we can take *any* function $\phi : D \to D$ and turn it into a function $e_Y \circ \phi \circ p_X : X \to Y$, and if we compose these processes we recover our original f:

$$e_Y \circ (p_Y \circ f \circ e_X) \circ p_X = (e_Y \circ p_Y) \circ f \circ (e_X \circ p_X) = f$$

Then we can actually *identify* any $f : X \to Y$ with its image $p_Y \circ f \circ e_X : D \to D$, and we get the following natural characterization:

$$\text{If } \phi : D \to D, \exists f : X \to Y. p_Y \circ f \circ e_X = \phi \iff (e_Y \circ p_Y) \circ \phi \circ (e_X \circ p_X) = \phi$$

The interesting thing to note here is that we've reduced the property involving multiple types to just the functions $(e_X \circ p_X), (e_Y \circ p_Y) : D \to D$, which we call c_X, c_Y. Since these are functions from D to itself, we can think of them as *untyped* functions. So if X is a retract of D, we can describe all functions involving X using D and a single function c_X.

Since c_X arises from a retraction, it has a special property, which is that it is *idempotent*: $c \circ c = c$. The intuition for this property comes from the section-retraction pairs. We think of the idempotent as a *representation* of a retract, identifying some "good" subset of a dynamic type by coercing all elements to behave like that type. Then we can identify X with the image of $c : D \to D$.

6.2 Universal Object: P_ω

So now that we've seen that retracts allow us to encode one type in another, we return to our goal, to construct a single type that characterizes some interesting class of types. This is called a *universal type* or in category terms, a *universal object.*

Definition 1. *A category has a universal object D if every object in the category has a section-retraction pair into D.*

One of the remarkable things that Dana Scott accomplished in Scott [1976] was that he constructed a universal countably-based continuous lattice: P_ω[3].

Without further ado, here it is:

$$P_\omega = \mathbb{N} \to \{\bot, \top\}$$

[3]pronounced "pee omega"

Surprised?

The idea is that an element of P_ω represents a subset of the natural numbers, and an element of a countably-based continuous lattice is determined by all of the observations you can make on it. So if you have a lattice X with countable basis b_i, then you can define the embedding as

$$e : X \to P_\omega$$

$$e(x) = \{i | b_i \sqsubseteq x\}$$

Where $b_i \sqsubseteq x$ essentially means that some finite observation b_i holds of x.

The projection, on the other hand, is less nice:

$$p : P_\omega \to X$$

$$p(S) = \bigsqcup \{b_i | i \in S\}$$

Where that $\bigsqcup$ corresponds operationally to a parallel exists, which is an impractical feature to have in your programming language (Plotkin [1977]).

While this universal type may be infeasible for a programming language implementation, the semantic properties it has are quite enticing! First, since every type is a retract of P_ω, we can understand all continuous functions by studying $P_\omega \to P_\omega$. More concretely, since retracts are determined by their idempotents, we can study constructions on *types* by functions on *terms*.

For example, instead of constructing non-well-founded recursive types by methods like D_∞, you can just use ordinary recursion on terms, much simpler!

Finally, while we can't use P_ω in our sequential languages, it turns out that many languages do have universal types. For a more thorough overview and much better exposition than here, I highly recommend Longley.

7 Translating Typed to Untyped: All Untyped Models are Unityped Models

Based on last section, we can turn the idea around, instead of taking our typed model and looking for an untyped model that "generates it", we might just start with our untyped language and see what "types" we can get from it.

This idea gets us to the main theorem of Scott [1980]:

Theorem 4. *Any model of untyped lambda calculus is embedded as a reflexive object in a cartesian closed category.*

That is, from a model of untyped lambda calculus, we can *construct* a cartesian closed category with a reflexive object such that the functions in the untyped model are exactly the endo-functions of the reflexive object.

Remembering the correspondence between typed lambda calculus and cartesian closed categories, *constructing* a CCC is the same as defining a *translation* of the typed lambda calculus, so the proof of this theorem is actually a translation from typed to untyped lambda calculus.

The intuition for the translation comes from the previous section: we want the types to be retracts of the dynamic type, so we will interpret the types as exactly the idempotent they represent on the dynamic type, which is then just an untyped lambda term!

So first, we interpret the types of the lambda calculus as untyped terms c such that $c(cx) \equiv cx$.

$$\begin{array}{rcl} [\![1]\!] & = & \lambda x.() \\ [\![A \times B]\!] & = & \lambda p.([\![A]\!](\pi_1 p), [\![B]\!](\pi_2 p)) \\ [\![A \to B]\!] & = & \lambda f.\lambda x.[\![B]\!](f([\![A]\!]x)) \end{array}$$

Which you may recognize as (except the 1) case, basically the corresponding contracts from Findler and Felleisen [2002]. I'll leave it as an exercise that these are actually idempotent.

Next, we want to translate typed terms to untyped terms, but in such a way that we maintain the validity of β, η equivalence. How can we achieve this? When you expand the domain of a function, you can call it on new inputs, so we have to make sure that these new inputs don't reveal any new information about our terms. That is, the equality for the untyped translation of a typed term should be completely determined by its behavior linking with other typed terms.

Fortunately, the interpretation of types above coerces *any* untyped term to behave like a term of a given type. So we just need to make sure that any interaction with a dynamically typed term is mediated by one of these idempotents.

It turns out that it's very simple to derive a translation that validates this equation, we just need to put idempotents on the variables!

$$\begin{array}{rcl} [\![x : A]\!] & = & [\![A]\!]x \\ [\![\lambda(x : A).t]\!] & = & \lambda x.[\![t]\!] \\ [\![tu]\!] & = & [\![t]\!][\![u]\!] \\ [\![(t, u)]\!] & = & ([\![t]\!], [\![u]\!]) \\ [\![\pi_i t]\!] & = & \pi_i [\![t]\!] \\ [\![()]\!] & = & () \end{array}$$

In order to prove preservation of η, we prove the following soundness theorem.

Theorem 5 (Soundness).
If $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$, then $x_1, \ldots, x_n \vdash [\![t]\!]$ and

$$[\![B]\!]([\![t]\!][([\![A_1]\!]x_1)/x_1 \cdots]) \equiv_\beta [\![t]\!]$$

equivalently,

$$[\![B]\!]([\![t]\!]) \equiv_\beta [\![t]\!] \equiv_\beta [\![t]\!][([\![A_1]\!]x_1)/x_1 \cdots]$$

Clearly the second formulation implies the first. To see the other direction, if we have $[\![B]\!]([\![t]\!])$, then

$$\begin{array}{rcl} [\![B]\!]([\![t]\!]) & \equiv_\beta & [\![B]\!]([\![B]\!]([\![t]\!][([\![A_1]\!]x_1)/x_1 \cdots])) \\ & \equiv_\beta & [\![B]\!]([\![t]\!][([\![A_1]\!]x_1)/x_1 \cdots]) \\ & \equiv_\beta & [\![t]\!] \end{array}$$

Where the second equation is from $[\![B]\!]$ being idempotent.

What does this have to do with validating η? Well if we look at the definitions of the $[\![A]\!]$s, we see that they are performing η expansion!

Theorem 6 (Soundness implies Eta Preservation)**.** *If Theorem* 5 *is true, then*

1. *If $t : A \to B$, then $\lambda x.[\![t]\!]([\![A]\!]x) \equiv_\beta [\![t]\!]$.*
2. *If $t : A \times B$, then $(\pi_1[\![t]\!], \pi_2[\![t]\!]) \equiv_\beta [\![t]\!]$.*
3. *If $t : 1$, then $() \equiv_\beta [\![t]\!]$.*

Proof. We'll do the first case, the others are similar. By soundness, $[\![A \to B]\!]([\![t]\!]) \equiv_\beta [\![t]\!]$, expanding the definition this means:

$$\begin{array}{rcl} \lambda x.[\![t]\!]([\![A]\!]x) & \equiv_\beta & \lambda x.([\![A \to B]\!][\![t]\!])([\![A]\!]x) \\ & \equiv_\beta & \lambda x.(\lambda y.[\![B]\!]([\![t]\!]([\![A]\!]y)))([\![A]\!]x) \\ & \equiv_\beta & \lambda x.[\![B]\!]([\![t]\!]([\![A]\!]([\![A]\!]x))) \\ & \equiv_\beta & \lambda x.[\![B]\!]([\![t]\!]([\![A]\!]x)) \\ & \equiv_\beta & [\![A \to B]\!][\![t]\!] \\ & \equiv_\beta & [\![t]\!] \end{array}$$

where the fourth equality is from idempotence of $[\![A]\!]$. □

So this shows that t is compiled in such a way that all interactions are protected by the idempotents $[\![A_i]\!], [\![B]\!]$.

So it remains to prove the soundness theorem

Proof. By induction on t. We do a few illustrative cases

1. $t = x : A$, then

$$\begin{array}{rcl} ([\![A]\!]([\![A]\!]x))[([\![A]\!]x)/x] & = & [\![A]\!]([\![A]\!]([\![A]\!]x)) \\ & = & [\![A]\!]([\![A]\!]x) \\ & = & ([\![A]\!]x) \end{array}$$

2. $t = \lambda(x : A).u$, then

$$\begin{array}{rcl} [\![A \to B]\!](\lambda x.[\![u]\!])[\cdots] & = & \lambda y.[\![B]\!]((\lambda x.[\![u]\!][\cdots])([\![A]\!]y)) \\ & = & \lambda y.[\![B]\!][\![u]\!][([\![A]\!]y)/x, \cdots] \\ & = & \lambda y.[\![u]\!][y/x] \\ & = & [\![\lambda(x : A).u]\!] \end{array}$$

3. $t = t'u$, then

$$\begin{array}{rcl} [\![B]\!]([\![t']\!][\![u]\!])[\cdots] & = & [\![B]\!]([\![t']\!][\cdots][\![u]\!][\cdots]) \\ & = & [\![B]\!]([\![t']\!][\![u]\!]) \\ & = & [\![B]\!](([\![A \to B]\!][\![t']\!])[\![u]\!]) \\ & = & [\![B]\!]((\lambda x.[\![B]\!]([\![t']\!]([\![A]\!]x)))[\![u]\!]) \\ & = & [\![B]\!]([\![B]\!]([\![t']\!]([\![A]\!][\![u]\!]))) \\ & = & [\![B]\!]([\![t']\!]([\![A]\!][\![u]\!])) \\ & = & (\lambda x.[\![B]\!]([\![t']\!]([\![A]\!]x)))[\![u]\!] \\ & = & ([\![A \to B]\!][\![t']\!])[\![u]\!] \\ & = & [\![t']\!][\![u]\!] \end{array}$$

□

8 Looking Forward

What can we glean today from Dana Scott's semantic insights?

Personally, I'm interested in gradual typing and I've found the ideas in Scott [1980] very illuminating in this regard. The hope is that by taking a bit more abstract of a perspective, we can prove some general results for constructing gradually typed languages. For example, the category constructed in section 7 is called the "Cauchy completion" or "Karoubi envelope" and this abstract perspective can be used to show that the construction of idempotents always works to validate η laws. Susumu Hayashi has an interesting paper about this idea Hayashi [1985], and I think it goes a long way to explaining why gradual typing has extended to so many language features.

References

Amal J Ahmed. *Semantics of types for mutable state.* PhD thesis, 2004.

Andrew W Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2001.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN Notices*, 2002.

Susumu Hayashi. Adjunction of semifunctors: categorical structures in nonextensional lambda calculus. *Theoretical computer science*, 1985.

Joachim Lambek. Deductive systems and categories i. syntactic calculus and residuated categories. *Mathematical Systems Theory*, 2(4), 1968.

Joachim Lambek. Deductive systems and categories ii. standard constructions and closed categories. *Category theory, homology theory and their applications I*, 1969.

Joachim Lambek. Deductive systems and categories iii. cartesian closed categories, intuitionist propositional calculus, and combinatory logic. In *Toposes, algebraic geometry and logic*. 1972.

Joachim Lambek. From lambda calculus to cartesian closed categories. 1980.

Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

F William Lawvere. Adjointness in foundations. *dialectica*, 23(3-4):281–296, 1969.

John R. Longley. *Universal Types and What They are Good For.*

Andrew M Pitts. Relational properties of domains. *Information and computation*, 1996.

Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical computer science*, 1977.

Dana Scott. Continuous lattices. In *Toposes, algebraic geometry and logic*, pages 97–136. Springer, 1972.

Dana Scott. Data types as lattices. *SIAM Journal on computing*, 5(3):522–587, 1976.

Dana Scott. Relating theories of the lambda calculus. 1980.

Dana S Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theoretical Computer Science*, 121(1-2):411–440, 1993.

Michael B Smyth and Gordon D Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 1982.

Mitchell Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 1979.