# Simple Typed Lambda Calculus

## 1 Sources of non-termination

We saw that the lambda calculus has many non-trivial non-terminating terms. We also saw that termination is undecidable in lambda calculus. But that shouldn't stop us from trying! Let's do something overkill instead. Instead of trying to decide termination perfectly, let's see if we can pare down lambda calculus to just a terminating fragment.

Let's start with the simple non-terminating term we saw before. that is,

$$R = (\lambda x.\ xx)(\lambda x.\ xx)$$

Is there something suspicious about this term? I'd say that generally $xx$ is a suspicious shape, and This term has three appearances of this pattern! So instead of the original

$$e_1, e_2 \in \Lambda \to e_1\, e_2 \in \Lambda$$

we should be a bit more discerning. Since our goal is to ultimately prove everything has a normal form, we can think about what that proof might look like. We saw last time that being able to show there is some function

$$f : \Lambda \to \mathbb{N} \quad f(e_1[x := e_2]) < f(e_1\, e_2)$$

is ideal. That is, a way to syntactically tell how many more substitutions we'll have to make to get to our normal form. Merely counting the applications would be great, but it doesn't actually satisfy this since the substitution can duplicate terms. So we'll need something slightly more nuanced, but honestly not much.

## 2 Types: summarizing terms

What's a type? A type system is a triple of a target language, a set of 'types terms', and a relation from terms (plus contexts) to types. The types are a set $\mathbf{T}$

1. $N \in \mathbf{T}$ (Just a group type, pick whatever you want)
2. $t_1, t_2 \in \mathbf{T}$ implies $t_1 \to t_2 \in \mathbf{T}$

And we will define the annotated lambda terms $\mathbf{A}$ as

1. $0, 1, ... \in \mathbf{A}$ (or whatever constants you want)
2. $x, y, ... \in \mathbf{A}$
3. $e \in \mathbf{A}, t \in \mathbf{T}$ implies $\lambda x : t.\, e \in \mathbf{A}$
4. $e_1, e_2 \in \mathbf{A}$ implies $e_1\, e_2 \in \mathbf{A}$

So far we have not done anything useful. We added some explicit constants, and type annotations. But haven't put any restrictions yet. I haven't even told you what the terms of the simply typed lambda calculus are yet!

To do that, we need to describe how a typing judgement works. A context $\Gamma$ is a finite subset of $V \times \mathbb{T}$ where each variable appears at most once. Then a judgement is, formally, a relation on the triple of contexts, terms, and types denoted $\vdash$ and used like $\Gamma \vdash x : t$. We typically define our $\vdash$ inductively like

$$\text{Cxt}\,\frac{(x,t) \in \Gamma}{\Gamma \vdash x : t}$$

The top line are assumptions, and the bottom are conslusion. Formally, this rule means

$$\{(\Gamma, x, t) \mid (x,t) \in \Gamma\} \subset \vdash$$

but nobody actually treats $\vdash$ as a set like this in the literature. I feel like a monstor for writing this at all.

The rules for simply typed lambda calculus are Cxt above, and

$$\text{Const}\,\frac{}{\Gamma \vdash n : N} \qquad \text{App}\,\frac{\Gamma \vdash x : t_1 \to t_2 \quad \Gamma \vdash y : t_1}{\Gamma \vdash x\, y : t_2} \qquad \text{Abs}\,\frac{\Gamma, (x, t_1) \vdash y : t_2}{\Gamma \vdash \lambda x : t_1.\, y : t_1 \to t_2}$$

First, some small thoughts. The empty precondition in the Const rule just means that judgement is always valid. The App rule has two preconditions. This is not unusual, so get used to that. Also, the Abs rule extends the context in the precondition. This is basically why we use a context, so we have a place to record function argument names and types.

## 3 Well Typed Terms

We say a term $x$ is well-typed (in the empty context) when there is a type $t$ where $\vdash x : t$. What does it mean really? It's often convenient to represent 'evidience of well-typedness' as a tree.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(x:N) \in (x:N),(y:N)}}{(x:N),(y:N) \vdash x : N \to N}}{(x:N) \vdash (\lambda y : N, x) : N \to N}}{\vdash (\lambda x : N, (\lambda y : N, x)) : N \to N \to N} \quad \dfrac{}{\vdash 1 : N}}{\vdash (\lambda x : N, (\lambda y : N, x))\ 1 : N \to N} \quad \dfrac{}{\vdash 5 : N}}{\vdash (\lambda x : N, (\lambda y : N, x))\ 1\ 5 : N}$$

Trees grow up in type theory. Here is such a tree that witnesses the well-typedness of the term at the bottom. So, a term is well-typed when it has a proof tree that ends in empty lines on all branches.

First, we define the Simply Typed Lambda Calculus (STLC) to be the rewrite system where, the terms are the well-typed members of the annotated lambda calculus, and its rewrite rule is $\beta'$ (trivially changed to accomidate the annotations by ignoring them).

For this to be sane, we need to verify that if

$$e_1 \underset{\beta'}{\longrightarrow} e_2$$

And $e_1$ is well-typed, then so is $e_2$. Unpacking a bit, this boils down to

$$\vdash (\lambda(x : t).\, e_1)e_2 : u \text{ implies } \vdash e_1[x := e_2] : u$$

Further unpacking yields the relevent lemma

$$(x : t) \vdash e_1 : u \wedge \vdash e_2 : t \text{ implies } \vdash e_1[x := e_2] : u$$

which is a trivial induction

Furthermore, this rewrite system is confluent since it embeds in lambda calculus. Annoyingly we added these annotations, so it's not literally just an application of confluence for lambda calculus. But it is pretty simple, and left as an exercise.

## 4 Judgements as a rewrite system

Here's where things get fun. A type system is nothing but a helper language associated with the core language. We have spent the last few weeks discussing how to ask good questions about language designs. So we can think about judgements themselves as a kind of rewrite system! But we have to be careful due to the having more than one precondition sometimes. So we can thing of it as a rewrite system on finite sets of judgements. Or we can think of it as rewrite systems on proof trees. We will formalize it the first way, but the visual of the tree is extremely useful.

We have a standard set of questions to ask about rewrite systems. Is it confluent? Is it normalizing?

### 4.1 Confluence

We will prove that the simply typed lambda calculus's has the diamond property. That is if $a \to b$ and $a \to c$ then there exists a $d$ with $b \to d$ and $c \to d$ all in one step.

The key observation is that there's a nice relationship between the recursive definition of the annotated lambda terms, and the judgements. That is, given a judgement $\Gamma \vdash x : t$, we know that $x$ has only one of four possible forms. For each form, there is exactly one judgement that can apply! This **syntax directed reasoning** is part of what makes type theory so useful.

So consider a set

$$a = \{\Gamma_1 \vdash x_1 : t_1, ..., \Gamma_n \vdash x_n : t_n\}$$

and suppose $a \to b$ and $a \to c$. Note each rule applies to only one element at a time. So we can consider $i, j$ so $a \to b$ affected only $i$ and $a \to c$ affected only $j$. If $i = j$, then exactly one rule could apply to $\Gamma_i \vdash x_i : t_i$ based on the variant of $x_i$. So both $a \to b$ and $a \to c$ are applications of the same rule. So $b = c$.

If $i \neq j$, then $b$ still has $\Gamma_j \vdash x_j : t_j$ as an element, and $c$ still has $\Gamma_i \vdash x_i : t_i$. So applying the unique rule for $j$ to $b$ and likewise for $i$ and $c$, we get confluence. From a proof tree perspective, if I do work on branch $A$, then work on branch $B$, that's the same as doing work on $B$ then $A$.

So now we know that this rewrite system is confluent. This does not guarantee that there is a unique type for each term! This turns out to be true, but we will deal with it later.

### 4.2 Normaliztion

The next thing to ask is about normal forms. We haven't excluded the possibility of some proof tree constructions repeating themselves an getting caught in a loop. As usual we want to count something decreasing with each rewrite. In this case, thankfully, there's an easy one. The complexity of the term decreases with each rule. And the tree has branching factor of two at worst. So for a term $x$ with complexity $n$, we have $2^n$ many rewrites at worst to build a proof tree for it.

So given a term, a context, and a type, deciding if $\Gamma \vdash x : t$ is easy to decide. Construct the proof tree, and see if it ends in blanks. Since the type system is confluent and strongly normalizing, this is a complete decision procedure! But it's not the one we want, since we it requires the type to be given.

We need a way to start with a term, and figure out its (unique) type. There's a nuanced way, where we build the tree up from its terminal leaves. And there's a brute force way. Let's do that instead. For a given $n$, observe that there are only finitely many types with less than $n$ arrows. Each in a valid proof tree either adds an annotated $t$, or adds one. So the size of the possible types of a term $x$ is bounded by

$$\text{lambdas in x} + \sum_{t \text{ in x}} \text{arrows in t}$$

Then we can scan all possible types, and we finally have a decision procedure for a term being a valid STLC term. Note there are better ones.

## 4.3 Remaining work

Great, so we've defined a nice fragment of the (annotated) lambda calculus. So what?

1. Does it actually eliminate our original target $\lambda x.\, xx$
2. Are there other bad terms it doesn't eliminate?
3. These annotations are an annoyance to write.
4. The choice of ground type seems arbitrary

Let's answer Question 1 now, and leave the rest for later. Can we give this thing a type? Suppose there was an annotation $t$ which made $\lambda(x:t).\, xx$ well-typed. Then for some $u$ we build the following

$$\dfrac{\dfrac{(x:t)\vdash x:t\to t \qquad \dfrac{\dfrac{}{(x:t)\vdash x:t}}{(x:t)\vdash xx:t}}{(x:t)\vdash xx:t}}{\vdash \lambda(x:t).\, xx:u}$$

This left branch has no more moves, since the only possible continuation is

$$\text{Nonsense!}\quad \dfrac{(x,t\to t)\in(x,t)}{(x:t)\vdash x:t\to t}$$

Since things are confluent, this is the only possible proof tree for $\vdash \lambda(x:t).\, xx:u$. So there is no such $t$ which makes this well-typed.

So by attaching this type system to lambda calculus, we have removed at least some of the scary terms we wanted.