Mini Class on Normalization in Type Theory

The Iowa Type Theory Commute

Aaron Stump
Computer Science
The University of Iowa

October 8, 2021

Contents

1 Introduction

The issue of whether or not programs terminate is of significant theoretical and practical importance in Computer Science. We will study one aspect of this topic, namely the normalization property for several different typed lambda calculi. Lambda calculus is a minimalistic programming language, whose central feature of anonymous functions is now found in most mainstream languages. We will review this below. Functional programming languages like LISP, Scheme, Haskell, and the various forms of ML (like OCaml) are based on lambda calculus, with Haskell and ML using statically typed versions.

Typed lambda calculus is also the core of proof assistants based on constructive type theory, like Coq and Agda. To ensure logical soundness of the theories implemented by these tools, it is necessary to ensure that all programs terminate. This is because, under the Curry-Howard isomorphism, proofs are identified with programs. For example, proofs by induction are identified with recursive programs: invoking the induction hypothesis in the proof is the same as making a recursive call. But then potentially diverging programs correspond to potentially unsound inductions, and must be banned by the system. Otherwise, one could prove any formula using a diverging function as the proof.

In this short class, we will see proofs of normalization properties for several different type theories, namely simply typed lambda calculus (STLC), System F (polymorphic lambda calculus), and System F_ω (a generalization of System F). These theories are described below. Let us begin with some basic definitions for typed lambda calculi.

2 Basics of Untyped Lambda Calculus

Lambda calculus expressions are called terms. The syntax for terms t is:

$$\textit{terms}\ t \quad ::= \quad x \mid t\ t' \mid \lambda\ x.\, t$$

Here, x is for variables, $t\ t'$ is for **applications** of t as a function to t' as an argument, and $\lambda\ x.\, t$ is a **lambda-abstraction**, an anonymous function which takes input x and returns output t. The λ in $\lambda\ x.t$ is said to bind x in t. It introduces local variable x within the **body** t of the lambda abstraction.

(Note that we are here following the common practice of introducing a specific family of meta-variables – namely t and variants like t', t_1, etc. – to refer to a specific set of mathematical objects, in this case terms. Anywhere you see a t, you should understand that it is referring to some term as defined by the above syntax.)

2.1 β-reduction

To define how programs (terms) of lambda calculus run, the central idea is so-called β-reduction. It says that an anonymous function applied to an argument can be reduced by substituting the argument for the bound variable in the body of the function. This basic idea is expressed by defining a binary relation $\leadsto$ on terms by this rule:

$$\frac{}{(\lambda\ x.\, t)\ t' \ \leadsto\ [t'/x]t}\ \beta$$

$[t'/x]t$ denotes the term one gets by capture-avoiding substitution of t' for x in t. Capture-avoiding means that the scoping of variables should not change when doing the substitution. So substituting $x\ x$ for y in $\lambda\ x.\, y$, for example, should <u>not</u> give us $\lambda x.\, (x\ x)$, because the scoping of x in $x\ x$ is global, but would become local if the result were $\lambda x.\, (x\ x)$. Instead, one may rename the bound variable to avoid such capture, giving a result like $\lambda y.\, (x\ x)$ instead. It is actually rather tricky to get the details for substitution exactly right, both in theory and in practice. We will content ourselves with the idea that we must rename bound variables to avoid capture (even though this means that strictly speaking, substitution is now not a function, since more than one renaming is possible).

Terms which are the same if one safely (i.e., without capture) renames bound variables are called α-equivalent.

Full β-reduction is the relation that allows the above β-reduction step to take place anywhere in a term. Since there can be more than *β-redex* (a term of the form $(\lambda\ x.\, t)\ t'$) in a term, full β-reduction is non-deterministic: it can happen that $t \leadsto t_1$ and $t \leadsto t_2$, where t_1 and t_2 are different. If for some similar relation we instead have that t_1 always equals t_2 in this situation, then the relation is *deterministic.* We will write $\leadsto^*$ for the reflexive-transitive closure of $\leadsto$. We have $t \leadsto^* t'$ iff there is a finite sequence of $\leadsto$-steps leading from t to t'. In other words, we allow 0 or more steps of β-reduction.

2.2 Example of β-reduction

Let *id* abbreviate the term $\lambda\ x.\, x$, and if n is a natural number ({0,1,...}), let $\dot{n}$ abbreviate

$$\lambda\ s.\, \lambda\ z.\ \underbrace{(s \cdots (s}_{n}\ \ z) \cdots)$$

This is the Church-encoding of number n. Then for example, we have

$$\dot{n}\ id \leadsto^* id$$

This is because the whole term $\dot{n}\ id$ itself is a β-redex (lambda abstraction applied to argument), which reduces to

$$\lambda\ z.\ \underbrace{(id \cdots (id}_{n}\ z) \cdots)$$

Reducing (sequentially) the n applications of id results in $\lambda\ z.\,z$, which is indeed α-equivalent to id. This is an artificial example, but one can define the usual arithmetic operations like addition, multiplication, etc., using the Church encoding.

2.3 Call-by-name reduction

Functional programming languages generally do not implement full β-reduction for their lambda calculus fragments, but rather some **reduction strategy**. This is a particular choice, for each term, of a subset of its β-redexes that may be reduced next. Below, we will show some of our normalization results for a strategy called **call-by-name**. This is a form of lazy reduction, where we do not require arguments t' to be reduced before reducing $(\lambda\ x.,t)\ t'$. We can define single-step call-by-name reduction with these inference rules:

$$\frac{}{(\lambda\ x.\,t)\ t' \ \rightsquigarrow\ [t'/x]t}\ \beta \qquad \frac{t_1 \ \rightsquigarrow\ t_1'}{t_1\ t_2 \ \rightsquigarrow\ t_1'\ t_2}\ \mathit{left}$$

Rather than just stipulating, as we did for full β-reduction, that any redex we find can be reduced, we here require a finite derivation of $t \rightsquigarrow t'$ using these rules. The first rule is just the β rule again, of course. The second, here called *left*, quite restricts where that β rule can be applied. It can only be applied in the left part of an application (or, using the β rule directly, at the top level of the term). A derivation built using these two rules has a use of the β-rule at the sole leaf of tree, followed by some sequence of uses of the *left*-rule. Here is an example derivation:

$$\frac{\dfrac{}{(\lambda\ x.\,\lambda\ y.\,\lambda z.\,(y\ x))\ (id\ id)\ \rightsquigarrow\ (\lambda\ y.\,\lambda\ z.\,(y\ (id\ id)))}\ \beta}{(\lambda\ x.\,\lambda\ y.\,\lambda z.\,(y\ x))\ (id\ id)\ id\ \rightsquigarrow\ (\lambda\ y.\,\lambda\ z.\,(y\ (id\ id)))\ id}\ \mathit{left}$$

This derivation proves the single reduction step (at the bottom of the derivation)

$$(\lambda\ x.\,\lambda\ y.\,\lambda z.\,(y\ x))\ (id\ id)\ id\ \rightsquigarrow\ (\lambda\ y.\,\lambda\ z.\,(y\ (id\ id)))\ id$$

We can write a call-by-name *reduction sequence* beginning with this step:

$$(\lambda\ x.\,\lambda\ y.\,\lambda z.\,(y\ x))\ (id\ id)\ id\ \rightsquigarrow\ (\lambda\ y.\,\lambda\ z.\,(y\ (id\ id)))\ id\ \rightsquigarrow\ \lambda\ z.\,(id\ (id\ id))$$

The second reduction also has a derivation, just using the β-rule. The final term of the reduction sequence – i.e., $\lambda\ z.\,(id\ (id\ id))$ – is a *normal form* with respect to call-by-name reduction: it cannot be reduced further with this relation. This is because the rules do not allow reduction underneath a λ-abstraction or in the right part of an application, since the only rule for finding a β-redex is *left*. This term is not a normal form with respect to full β-reduction, however, because it has two β-redexes in it, namely the two applications of *id*. This shows that normalization with call-by-name reduction might result in a term that has β-redexes in it, in the body of a λ-abstraction. It could also happen, if one has free (undeclared) variables in the term, that there could be a β-redex in the right part of an application of a variable, as in $x\ (id\ id)$. But if the term is **closed** (no occurrences of variable x except under some λ that binds x), then terms which are normal with respect to call-by-name reduction are λ-abstractions. These might contain β-redexes in their bodies.

Call-by-name reduction is a deterministic strategy.

3 Normalization for the Simply Typed Lambda Calculus

In this section, we will prove that terms that are statically typable in a particular type system called the Simply Typed Lambda Calculus (STLC) are normalizing with respect to full β-reduction. Since β-reduction is nondeterministic, an important issue is whether we will prove that no matter which redex one chooses to reduce next, reduction reaches a normal form; or instead that there exists a particular choice of redex that guarantees reduction reaches a normal form (but where it is conceivable that reduction could diverge if a different strategy were applied). Reduction no matter what strategy is used is called **strong normalization** (also **termination**), while reduction under some strategy (but possibly not all) is called **weak normalization**, or just normalization. Here we will prove just weak normalization.

3.1 Simply typing

STLC is a basic type system that one finds as a subsystem of most practical type systems. It is thus of interest. Type systems always come with some set of types. The simple types T are defined over some unspecified set of base types (in practice, these could be types like `char` or `int`):

$$\begin{array}{rcl} \textit{base types } b & & \\ \textit{simple types } T & ::= & b \mid T_1 \to T_2 \end{array}$$

A type system specifies when terms can be assigned types. For example, intuitively, the term id can be assigned any type of the form $T \to T$; for example, $b \to b$.

Type systems are usually presented using a set of inference rules

4 Call-by-name Normalization for System F

This section gives a proof that call-by-name reduction is normalizing for unannotated System F (polymorphic lambda calculus), and considers a few consequences. System F is defined with annotated terms, where λ-bound variables must be declared with their types. So we have $\lambda x : T.t$ instead of just $\lambda x.t$. For metatheoretic analysis, I prefer to work with unannotated terms. This system (with unannotated terms) is also called $\lambda 2$.

5 Syntax

$$\begin{array}{lcl} \textit{term variables } x & & \\ \textit{type variables } X & & \\ \textit{terms } t & ::= & x \mid \lambda x.t \mid t\ t' \\ \textit{types } T & ::= & X \mid T \to T' \mid \forall X.T \end{array}$$

6 Typing

A typing context Γ declares free term and type variables:

$$\textit{Typing context } \Gamma ::= \cdot \mid \Gamma, x : T \mid \Gamma, X : \star$$

We treat Γ as a function, and write $\Gamma(x) = T$ to mean that Γ contains a declaration $x : T$. We will implicitly require that Γ does not declare any variable x twice. Variables can be implicitly renamed in λ-terms to

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x.t : T \to T'} \qquad \frac{\Gamma \vdash t : T_1 \to T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t\ t' : T_2}$$

$$\frac{\Gamma, X : \star \vdash t : T}{\Gamma \vdash t : \forall X.T} \qquad \frac{\Gamma \vdash t : \forall X.T \quad \Gamma \vdash T' : \star}{\Gamma \vdash t : [T'/X]T}$$

Figure 1: Typing rules for unannotated System F

$$\frac{\Gamma(X) = \star}{\Gamma \vdash X : \star} \qquad \frac{\Gamma \vdash T_1 : \star \quad \Gamma \vdash T_2 : \star}{\Gamma \vdash T_1 \to T_2 : \star} \qquad \frac{\Gamma, X : \star \vdash T : \star}{\Gamma \vdash \forall X.T : \star}$$

Figure 2: Kinding rules for unannotated System F

make it possible to enforce this requirement. The typing rules are in Figure 1. To ensure that types are well-formed, we use some extra rules, called *kinding* rules, in Figure 2.

7 Semantics for types

Figure 3 gives a compositional semantics $[\![T]\!]_\rho$ for types. The function ρ gives the interpretations of free type variables in T. Each free type variable is interpreted as a *reducibility candidate*, and write ρ only for functions mapping type variables X to reducibility candidates. To define what a reducibility candidate is: let us denote the set of closed terms which normalize using call-by-name reduction as $\mathcal{N}$. We will write $\rightsquigarrow$ for call-by-name reduction. Then a reducibility candidate R is a set of terms satisfying the following requirements:

- $R \subseteq \mathcal{N}$
- If $t \in R$ and $t' \rightsquigarrow t$, then $t' \in R$

The set of all reducibility candidates is denoted $\mathcal{R}$.

Lemma 1 ($\mathcal{R}$ is a cpo)**.** *The set $\mathcal{R}$ ordered by subset forms a complete partial order, with greatest element $\mathcal{N}$ and greatest lower bound of a nonempty set of elements of $\mathcal{R}$ given by intersection.*

Proof. $\mathcal{N}$ satisfies both requirements for a reducibility candidate, and since one of those requirements is being a subset of $\mathcal{N}$, it is clearly the largest such set to do so. Let us prove that the intersection of a nonempty set S of reducibility candidates is still a reducibility candidate. Certainly if the members of S are subsets of $\mathcal{N}$ then so is $\bigcap S$. For the second property: assume an arbitrary $t \in \bigcap S$ with $t' \rightsquigarrow t$, and show $t' \in \bigcap S$. For the latter, it suffices to show $t' \in R$ for every $R \in S$. Consider an arbitrary such R. From $t \in \bigcap S$ and $R \in S$, we have $t \in R$. Then since R is a reducibility candidate, $t \in R$ and $t' \rightsquigarrow t$ implies $t' \in R$, . $\square$

$$\begin{array}{lcl}
[\![X]\!]_\rho & = & \rho(X) \\
[\![T_1 \to T_2]\!]_\rho & = & \{t \in \mathcal{N} \mid \forall t' \in [\![T_1]\!]_\rho.\ t\ t' \in [\![T_2]\!]_\rho\} \\
[\![\forall X.T]\!]_\rho & = & \bigcap_{R \in \mathcal{R}} [\![T]\!]_{\rho[X \mapsto R]}
\end{array}$$

Figure 3: Reducibility semantics for types

Lemma 2 (The semantics of types computes reducibility candidates)**.** *If $\rho(X)$ is defined for every free type variable of T, then $[\![T]\!]_\rho \in \mathcal{R}$.*

Proof. The proof is by induction on the structure of the type. If T is a type variable X, then by assumption, $\rho(X)$ is a reducibility candidate, and this is the value of $[\![T]\!]_\rho$.

If T is an arrow type $T_1 \to T_2$, we must prove the two properties listed above for being a reducibility candidate. Certainly $[\![T]\!]_\rho \subseteq \mathcal{N}$, because the semantics of arrow types requires this explicitly. Now suppose that $t \in [\![T_1 \to T_2]\!]_\rho$ and $t' \rightsquigarrow t$. We must show $t' \in [\![T_1 \to T_2]\!]_\rho$. Since t is normalizing and $t' \rightsquigarrow t$, we know that t' is also normalizing (there is a reduction sequence from t' to t and from t to a normal form). So let us assume an arbitrary $t'' \in [\![T_1]\!]_\rho$, and show that $t'\ t'' \in [\![T_2]\!]_\rho$. Since $t' \rightsquigarrow t$, by the definition of call-by-name reduction, we have

$$t'\ t'' \rightsquigarrow t\ t''$$

Since $t \in [\![T_1 \to T_2]\!]_\rho$, we know by the semantics of types that $t\ t'' \in [\![T_2]\!]_\rho$, since $t'' \in [\![T_1]\!]_\rho$. By the IH, $[\![T_2]\!]_\rho$ is a reducibility candidate. So since $t'\ t'' \rightsquigarrow t\ t''$ and $t\ t'' \in [\![T_2]\!]_\rho$, we also have $t'\ t'' \in [\![T_2]\!]_\rho$. This was all we had to prove in this case.

Finally, if T is a universal type $\forall X.T'$, then by IH, the set $[\![T']\!]_{\rho[X\mapsto R]}$ is a reducibility candidate for all $R \in \mathcal{R}$. Since $\mathcal{R}$ is a complete partial order, $\bigcap_{R\in\mathcal{R}} [\![T']\!]_{\rho[X\mapsto R]}$ is then also a reducibility candidate.

□

8 Soundness of Typing Rules

The goal of this section is to prove that terms which can be assigned a type using the rules of Figure 1 are normalizing. We will actually prove a stronger statement, based on an interpretation of typing judgments. First, we must define an interpretation $[\![\Gamma]\!]$ for typing contexts Γ. This interpretation will be a set of pairs (σ, ρ), where ρ is, as above, a function mapping type variables to reducibility candidates; and σ maps term variables to terms. The definition is by recursion on the structure of Γ:

$$\begin{array}{lcl} (\sigma,\rho) \in [\![x : T, \Gamma]\!] & \Leftrightarrow & \sigma(x) \in [\![T]\!]_\rho \ \wedge\ (\sigma,\rho) \in [\![\Gamma]\!] \\ (\sigma,\rho) \in [\![X : *, \Gamma]\!] & \Leftrightarrow & \rho(x) \in \mathcal{R} \ \wedge\ (\sigma,\rho) \in [\![\Gamma]\!] \\ (\sigma,\rho) \in [\![\cdot]\!] & & \end{array}$$

In the statement of the theorem below, we write σt to mean the result of simultaneously substituting $\sigma(x)$ for x in t, for all x in the domain of σ.

Lemma 3. *Suppose $(\sigma,\rho) \in [\![\Gamma]\!]$. If $t \in [\![T]\!]_\rho$, then $(\sigma[x \mapsto t], \rho) \in [\![\Gamma, x : T]\!]$. Also, if $R \in \mathcal{R}$, then $(\sigma, \rho[x \mapsto R]) \in [\![\Gamma, X : *]\!]$.*

Proof. The proof of the first part is by induction on Γ. If $\Gamma = \cdot$, then to show $(\sigma[x \mapsto t], \rho) \in [\![\cdot, x : T]\!]$, it suffices to show $t \in [\![T]\!]_\rho$, which holds by assumption. If $\Gamma = y : T', \Gamma'$, then we have $(\sigma,\rho) \in [\![\Gamma']\!]$ by the definition of $[\![\Gamma]\!]$, and we may apply the IH to conclude $(\sigma[x \mapsto t], \rho) \in [\![\Gamma', x : T]\!]$, from which we can conclude the desired $(\sigma[x \mapsto t], \rho) \in [\![\Gamma, x : T]\!]$, again by the definition of $[\![\Gamma]\!]$. Similar reasoning applies if $\Gamma = X : \star, \Gamma'$. The proof of the second part of the lemma is exactly analogous. □

Theorem 4 (Soundness of typing rules with respect to the semantics)**.** *If $\Gamma \vdash t : T$, then for all $(\sigma,\rho) \in [\![\Gamma]\!]$, we have $\sigma t \in [\![T]\!]_\rho$.*

Proof. The proof is by induction on the structure of the assumed typing derivation. In each case, we will implicitly assume an arbitrary $(\sigma,\rho) \in [\![\Gamma]\!]$.

Case:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

We proceed by inner induction on Γ. If Γ is empty, then $\Gamma(x) = T$ is false, and this case cannot arise. Suppose Γ is of the form $x : T, \Gamma'$. Then $\sigma(x) \in [\![T]\!]_\rho$ by definition of $[\![\Gamma]\!]$, which suffices to prove the conclusion. Suppose Γ is of the form $y : T, \Gamma'$, where $y \neq x$, or of the form $X : *, \Gamma'$. Then $\Gamma'(x) = T$ and $(\sigma, \rho) \in [\![\Gamma']\!]$, and we use the induction hypothesis to conclude $\sigma x \in [\![T]\!]_\rho$.

Case:

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x.t : T \to T')}$$

To prove $(\lambda x.\sigma t) \in [\![T \to T']\!]_\rho$, it suffices to assume an abitrary $t' \in [\![T]\!]_\rho$ and prove $(\lambda x.\sigma t)\ t' \in [\![T']\!]_\rho$. Since $[\![T']\!]_\rho$ is a reducibility candidate, it suffices to prove $[t'/x]\sigma t \in [\![T']\!]_\rho$, since $(\lambda x.\sigma t)\ t' \rightsquigarrow [t'/x](\sigma t)$. But if we let $\sigma' = \sigma[x \mapsto t']$, then we have $(\sigma', \rho) \in [\![\Gamma, x : T]\!]$ by Lemma 3, so we may apply the IH to conclude $\sigma' t \in [\![T']\!]_\rho$, as required.

Case:

$$\frac{\Gamma \vdash t : T_1 \to T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t\ t' : T_2}$$

By the IH, $\sigma t \in [\![T_1 \to T_2]\!]_\rho$ and $\sigma t' \in [\![T_1]\!]_\rho$. By the semantics of arrow types, this immediately implies $(\sigma t)\ (\sigma t') \in [\![T_2]\!]_\rho$, as required.

Case:

$$\frac{\Gamma, X : \star \vdash t : T}{\Gamma \vdash t : \forall X.T}$$

We must prove $\sigma t \in [\![\forall X.T]\!]_\rho$. By the semantics of universal types, it suffices to assume an arbitrary $R \in \mathcal{R}$, and prove $\sigma t \in [\![T]\!]_{\rho[X \mapsto R]}$. But this follows by the IH, which we can apply because $(\sigma, \rho[X \mapsto R]) \in [\![\Gamma, X : \star]\!]$, by Lemma 3.

Case:

$$\frac{\Gamma \vdash t : \forall X.T \quad \Gamma \vdash T' : \star}{\Gamma \vdash t : [T'/X]T}$$

By the IH, we know $\sigma t \in [\![\forall X.T]\!]_\rho$, which by the semantics of universal types is equivalent to

$$\sigma t \in \bigcap_{R \in \mathcal{R}} T_{\rho[X \mapsto R]} \tag{1}$$

Since $(\sigma, \rho) \in [\![\Gamma]\!]$, we may easily observe that ρ is defined for all the free type variables of T'. So by Lemma 2, $[\![T']\!]_\rho \in \mathcal{R}$. From the displayed formula above (1), we can conclude $\sigma t \in [\![T]\!]_{\rho[X \mapsto [\![T']\!]_\rho]}$. Now we must apply the following lemma, whose easy proof by induction on T we omit, to conclude $\sigma t \in [\![[T'/X]T]\!]_\rho$.

Lemma 5. $[\![[T'/X]T]\!]_\rho = [\![T]\!]_{\rho[X \mapsto T']}$

□