The Objective Metatheory of Simply Typed Lambda Calculus

Astra Kolomatskaia *(she/her)*

The New York Category Theory Seminar

Lambda Calculus

Functions in Mathematics

We are familiar with defining a function by a formula:

$$f : \mathbb{R} \to \mathbb{R}$$
$$f\ (x) = x^2$$

The same definition may be made without assigning the name f:

$$x \mapsto x^2$$

In computer science and type theory, we use lambdas to denote the same:

$$\lambda x.\ x^2$$

Inductive Definitions

We can construct the natural numbers by saying that a `Nat` is either:

`Z` or `S` of another `Nat`

We can capture this definition using a grammar:

```
n = Z
  | S n
```

From this, we can conclude that `(S (S (S Z)))` is a `Nat`

Untyped Lambda Calculus

We construct a universe of functions defined by their formulas (terms)
Very solipsistically, these functions will act on other functions

`t = x`	*Variable*
`| λ x. t`	*Abstraction*
`| t t`	*Application*

Some examples of lambda terms:

a – an unbound variable
$\lambda x.\ x$ – the identity
$\lambda x.\ a$ – a constant function
$\lambda s.\ \lambda z.\ s\ (s\ z)$ – takes a function and argument and applies the function twice

Reduction

If $f\ (x) = x^2$, then $f\ (3) = 3^2$

The usual semantics of *functions defined by formulas* is that if you apply such a function to an argument, then this reduces to the body of the function, but with all occurrences of the variable replaced with the argument (substitution)

For lambda terms, we assert that:

$$(\lambda x.\ t)\ s \Rightarrow t\ [x \mapsto s]$$

This is known as the β law

An Example of Reduction

In the following trace, we repeatedly do *two steps at once*:

$(\lambda n.\ \lambda m.\ \lambda s.\ \lambda z.\ n\ s\ (m\ s\ z))\ (\lambda s.\ \lambda z.\ s\ (s\ z))\ (\lambda s.\ \lambda z.\ s\ (s\ z))$

$\Rightarrow\ \lambda s.\ \lambda z.\ (\lambda s.\ \lambda z.\ s\ (s\ z))\ s\ ((\lambda s.\ \lambda z.\ s\ (s\ z))\ s\ z)$

$\Rightarrow\ \lambda s.\ \lambda z.\ (\lambda s.\ \lambda z.\ s\ (s\ z))\ s\ (s\ (s\ z))$

$\Rightarrow\ \lambda s.\ \lambda z.\ s\ (s\ (s\ (s\ z)))$

This is a calculation that $2 + 2 = 4$ in the Church encoding of natural numbers

Normalisation

In the previous computational trace, the final expression

$$\lambda s.\ \lambda z.\ s\ (s\ (s\ (s\ z)))$$

was such that no further reduction rules applied to it

Such an expression is considered a normal form

On the other hand, consider the term:

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

Since it is a function abstraction being applied to an argument, the β law applies

However, this will repeatedly reduce to itself; there is no hope of normalisation!

Types

Minimal Logic

Let A, B, C, ... be a collection of uninterpreted atomic propositions

We can form compound propositions with implication $\rightarrow$, for example:

$$A \rightarrow B \rightarrow A$$

We then define what it would mean to prove such a proposition

In our proof systems we will have judgments of the form $\Gamma \vdash T$

Here, Γ is a list of assumptions (a context)

Inference Rules

The system of minimal logic has three proof rules:

$$\frac{T \in \Gamma}{\Gamma \vdash T}\ Ass$$

$$\frac{\Gamma, T \vdash S}{\Gamma \vdash T \to S}\ \to_I$$

$$\frac{\Gamma \vdash T \to S \qquad \Gamma \vdash T}{\Gamma \vdash S}\ \to_E$$

Two Derivations

It is best to read these from bottom to top:

$$\dfrac{\dfrac{\dfrac{}{A,\ B \vdash A}\ Ass}{A \vdash B \to A}\ \to_I}{\vdash A \to B \to A}\ \to_I$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A,\ A \to B \vdash A \to B}\ Ass \qquad \dfrac{}{A,\ A \to B \vdash A}\ Ass}{A,\ A \to B \vdash B}\ \to_E}{A \vdash (A \to B) \to B}\ \to_I}{\vdash A \to (A \to B) \to B}\ \to_I$$

Constructive Logic

The *BHK interpretation* of a proof of an implication $T \to S$ is a construction that, when given a proof of T, produces a proof of S

We will identify the *type* T with the *space* of proofs of T

A proof of $T \to S$, then, is a function from T to S

Given a proof f of $T \to S$, and a proof a of T, the application $f\ a$ is a proof of S

But we already have an excellent language for discussing functions and application!

Simply Typed Lambda Calculus

Syntax:

```
T = X
  | T → T

t = x
  | λ (x : T). t
  | t t
```

Here X ranges over *type variables* and x ranges over *term variables*

Expressions defined by T are types, and by t are terms

Typing Rules

Contexts will now consist of type annotations for variables

We augment our proof rules for minimal logic with term annotations:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\, Var$$

$$\frac{\Gamma,\ x : T \vdash t : S}{\Gamma \vdash \lambda\,(x : T).\ t : T \to S}\ \to_I$$

$$\frac{\Gamma \vdash f : T \to S \qquad \Gamma \vdash a : T}{\Gamma \vdash f\ a : S}\ \to_E$$

A term t is said to be well-typed in context Γ if we have $\Gamma \vdash t : T$ for some T

We only work with well typed terms!

Fundemental Metatheory

With the way that we have set up the theory, we have:

a] Every well typed term has a unique type

b] Every well typed term has a unique typing derivation

c] Every minimal logic derivation corresponds to an unique proof term
(up to renaming variables)

d] Typing derivations can be reconstructed by a structurally recursive algorithm

e] Computation preserves types

Proof Trees with Term Augmentations

$$\dfrac{\dfrac{\dfrac{}{a : A,\ b : B \vdash a : A}\ Var}{a : A \vdash \lambda\,(b : B).\ a : B \to A}\ \to_I}{\vdash\ \lambda\,(a : A).\ \lambda\,(b : B).\ a : A \to B \to A}\ \to_I$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A,\ f : A \to B \vdash f : A \to B}\ Var \qquad \dfrac{}{a : A,\ f : A \to B \vdash a : A}\ Var}{a : A,\ f : A \to B \vdash f\ a : B}\ \to_E}{a : A \vdash \lambda\,(f : A \to B).\ f\ a : (A \to B) \to B}\ \to_I}{\vdash\ \lambda\,(a : A).\ \lambda\,(f : A \to B).\ f\ a : A \to (A \to B) \to B}\ \to_I$$

As you can see, the annotations on variables can be recovered from the type
(at least as far as these examples are concerned)

Deep Metatheory

THEOREM (**Normalisation**):
Every well-typed term reduces to a normal form in a finite number of steps

COROLLARY:
The term $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ *is not well-typed (for any variable annotations)*

Objective Metatheory

Motivation

When we implement a programming language, how do we know that we haven't make a mistake in the definition of substitution or in handling names?

For that matter, if attempt to specify STLC on paper, we have a multitude of decisions to make: names or de Bruijn indicies, explicit or direct substitution, etc. *Are we talking about the same language?*

SOLUTION [Lambek 1980, known in folklore for ~10 years prior]:

Here is a one sentence definition of STLC:
a freely generated contextual cartesian closed category

Contexts and Indexed Lists

Given a type ty, we define Ctx_{ty} to be the inductive type generated by:

- an empty context $\varnothing : \mathrm{Ctx}_{ty}$, and
- for every $\Gamma : \mathrm{Ctx}_{ty}$ and $A : ty$, an extended context $\Gamma \dashv A : \mathrm{Ctx}_{ty}$

Given a family $el : ty \to \mathrm{Type}$, we define the family $\mathrm{Els}_{el} : \mathrm{Ctx}_{ty} \to \mathrm{Type}$ to be the indexed inductive type generated by:

- an empty indexed list $! : \mathrm{Els}_{el}\ \varnothing$, and
- for every $\sigma : \mathrm{Els}_{el}\ \Gamma$ and $t : el\ A$, a constructed indexed list $\sigma \oplus t : \mathrm{Els}_{el}\ (\Gamma \dashv A)$

Given a family $tm : \mathrm{Ctx}_{ty} \to ty \to \mathrm{Type}$, we define the family $\mathrm{Tms}_{tm} : \mathrm{Ctx}_{ty} \to \mathrm{Ctx}_{ty} \to \mathrm{Type}$ by $\mathrm{Tms}_{tm}\ \Gamma\ \Delta = \mathrm{Els}_{tm\ \Gamma}\ \Delta$

In general, things indexed by a context and a type will be called *terms*, and elements of $\mathrm{Tms}_{tm}\ \Gamma\ \Delta$ are called *substitutions*

Intrinsically Typed de Bruijn Variables

Given a type ty, we define the familly $\mathrm{Var}_{ty} : \mathrm{Ctx}_{ty} \to ty \to \mathrm{Type}$ to be the indexed inductive type generated by:

- a zero variable $zv : \mathrm{Var}_{ty}\ (\Gamma \dashv A)\ A$, and
- for every $v : \mathrm{Var}_{ty}\ \Gamma\ A$, a successor variable $sv\ v : \mathrm{Var}_{ty}\ (\Gamma \dashv B)\ A$

$\mathrm{Var}_{ty}\ \Gamma\ A$ is not a subtype of $\mathrm{Fin}\ _{\mathrm{length}\ \Gamma}$; the first constructor secretly hides something analogous to the refl constructor of a Martin-Löf identity type!

Variables can be used to index into indexed lists via the function $\mathrm{derive} : \mathrm{Els}_{el}\ \Gamma \to \mathrm{Var}_{ty}\ \Gamma\ A \to el\ A$

Renamings

Variables are indexed by a contest and type, and can thus be thought of as terms

Substitutions made of variables are known as *renamings*: $\mathrm{Ren}_{ty} \equiv \mathrm{Tms}\ \mathrm{Var}_{ty}$

We can define a category structure on renamings:

For $v : \mathrm{Var}_{ty}\ \Delta\ A$, $\sigma : \mathrm{Ren}_{ty}\ \Gamma\ \Delta$, define $v\ [\![\ \sigma\ ]\!] : \mathrm{Var}_{ty}\ \Gamma\ A$ by $\mathrm{derive}\ \sigma\ v$

For $\sigma : \mathrm{Ren}_{ty}\ \Delta\ \Sigma$, $\tau : \mathrm{Ren}_{ty}\ \Gamma\ \Delta$, define $\sigma \circledcirc \tau : \mathrm{Ren}_{ty}\ \Gamma\ \Sigma$ by $\mathrm{map}\ (-[\![\ \tau\ ]\!])\ \sigma$

For $A : ty$, $\sigma : \mathrm{Ren}_{ty}\ \Gamma\ \Delta$ define $\mathrm{W}_1\ A\ \sigma : \mathrm{Ren}_{ty}\ (\Gamma \div A)\ \Delta$ by $\mathrm{map}\ sv\ \sigma$, and $\mathrm{W}_2\ A\ \sigma : \mathrm{Ren}_{ty}\ (\Gamma \div A)\ (\Delta \div A)$ by $\mathrm{W}_1\ A\ \sigma \oplus zv$

Define $id\ \Gamma : \mathrm{Ren}_{ty}\ \Gamma\ \Gamma$ by repeatedly applying W_2 to $!$

We can also prove left and right identity laws, as well as associativity

Simple Contextual Categories

A *simple contextual category* consists of the following:

(i) a type ty of *types*,

▷ from which we derive the type $ctx \equiv \mathrm{Ctx}_{ty}$ of *contexts*,

(ii) for each $\Gamma : ctx$ and $A : ty$, a type $tm\ \Gamma\ A$ of *terms*,

▷ from which we derive, for each $\Gamma\ \Delta : ctx$, the type $tms\ \Gamma\ \Delta \equiv \mathrm{Tms}_{tm}\ \Gamma\ \Delta$ of *substitutions*

(iii) for each $t : \mathrm{Tms}\ \Delta\ A$ and $\sigma : \mathrm{tms}\ \Gamma\ \Delta$, a term $t\ [\![\ \sigma\ ]\!] : tms\ \Gamma\ A$

▷ from which we derive, for each $\sigma : tms\ \Delta\ \Sigma$ and $\tau : tms\ \Gamma\ \Delta$, the substitution $\sigma \circledcirc \tau \equiv \mathrm{map}\ (-[\![\ \tau\ ]\!])\ \sigma : \mathrm{tms}\ \Gamma\ \Delta$

(iv) for each $\Gamma : ctx$, a substitution $id\ \Gamma : tms\ \Gamma\ \Gamma$

Simple Contextual Categories *[cont.]*

The above are subject to the following laws:

(v) for every $\sigma : tms\ \Gamma\ \Delta$, we have $id\ \Delta \circledcirc \sigma = \sigma$

(vi) for every $t : tm\ \Gamma\ A$, we have $t\ [\![\ id\ \Gamma\ ]\!] = t$

- ▷ from which we derive, by applying the above in each component, that for every $\sigma : tms\ \Gamma\ \Delta$, we have $\sigma \circledcirc id\ \Gamma = \sigma$

(vii) for every $t : tm\ \Sigma\ A$, $\sigma : tms\ \Delta\ \Sigma$, and $\tau : tms\ \Gamma\ \Delta$, we have $t\ [\![\ \sigma\ ]\!]\ [\![\ \tau\ ]\!] = t\ [\![\ \sigma \circledcirc \tau\ ]\!]$

- ▷ from which we derive, by applying the above in each component, that for every $\sigma : tms\ \Sigma\ \Omega$, $\tau : tms\ \Delta\ \Sigma$, and $\mu : tms\ \Gamma\ \Delta$, we have $(\sigma \circledcirc \tau) \circledcirc \mu = \sigma \circledcirc (\tau \circledcirc \mu)$

(viii) each $tm\ \Gamma\ A$ is an h-set

- ▷ from which we derive that every $tms\ \Gamma\ \Delta$ is an h-set

Weakening Theory

Substitutions are indexed lists of terms; the first thing that we can do with a non-empty list is apply first and rest to it

We obtain $z : tm\ (\Gamma \dot{+} A)\ A$ and $\pi : tms\ (\Gamma \dot{+} A)\ \Gamma$ via $id\ (\Gamma \dot{+} A) \equiv \pi \oplus z$

For $B : ty$ and $t : tm\ \Gamma\ A$ define $\mathrm{W}_1\ A\ t : tm\ (\Gamma \dot{+} B)\ A$ by $t\ [\![\ \pi\ ]\!]$

We can prove that for $t : tm\ \Delta\ A$, $\sigma : tms\ \Gamma\ \Delta$, and $s : tm\ \Gamma\ B$, that $(W_1\ B\ t)\ [\![\ \sigma \oplus s\ ]\!] = t\ [\![\ \sigma\ ]\!]$

Variable Theory

More interestingly, we can use variables to index components of the identity, which we will also call *variables*

Let $var \equiv \mathrm{Var}_{ty}$, and for $v : var\ \Gamma\ A$, let $\mathrm{makeVar}\ v \equiv \mathrm{derive}\ (id\ \Gamma)\ v$

We are able to show that $(\mathrm{makeVar}\ v)\ [\![\ \sigma\ ]\!] = \mathrm{derive}\ \sigma\ v$

Via the following relation, $\mathrm{makeVar}\ (sv\ v) = W_1\ A\ (\mathrm{makeVar}\ v)$, relating variables to their successors, we are able to prove the following functoriality law $\mathrm{makeVar}\ (v\ [\![\ \sigma\ ]\!]) = (\mathrm{makeVar}\ v)\ [\![\ \mathrm{makeRen}\ \sigma\ ]\!]$

It is also easy to show that $\mathrm{makeRen}\ (id\ \Gamma) = id\ \Gamma$

This establishes that any contextual category is the target of a contextual structure preserving functor from an internal contextual (pre)category of renamings

Contextual CCCs

A simple contextual category is *cartesian closed* provided that it is endowed with the following structure:

(i) for every $A, B : ty$, an arrow type $A \Rrightarrow B$

(ii) for every $t : tm\ (\Gamma \dotplus A)\ B$, an abstraction $\Lambda\ t : tm\ \Gamma\ (A \Rrightarrow B)$

(iii) for every $t : tm\ \Gamma\ (A \Rrightarrow B)$ and $s : tm\ \Gamma\ A$, an application $app\ t\ s : tm\ \Gamma\ B$

▷ from which we derive, for every $t : tm\ \Gamma\ (A \Rrightarrow B)$, the categorical app $App\ t \equiv app\ (W_1\ A\ t)\ z : tm\ (\Gamma \dotplus A)\ B$

Subject to the following laws:

(iv) naturality of Λ: $(\Lambda\ t)\ [\![\ \sigma\ ]\!] = \Lambda\ (t\ [\![\ W_2\ A\ \sigma\ ]\!])$

(v) the β-law: $app\ (\Lambda\ t)\ s = t\ [\![\ id\ \Gamma\ \oplus\ s\ ]\!]$

(vi) the η-law: $t = \Lambda\ (App\ t)$

What is STLC?

To wrap up, we define the notion of a contextual functor, and what is means for a contextual functor between contextual CCCs to preserve the CCC structure

In the case of STLC with one base type, we work in the category of pointed contextual CCCs and CCC and basepoint preserving contextual functors

We then say that an *implementation of STLC* is an initial object of this category

MAIN RESULT: *every implementation of STLC admits a normalisation theorem, which, in particular, implies the decidable equality of terms*

Normalisation by Evaluation

STLC Syntax

In this section, we will work with a concrete syntax

We define STLC types Ty to be the inductive type generated by:

- A base type $\mathrm{Base} : \mathrm{Ty}$
- For every A, $B : \mathrm{Ty}$, an arrow type $A \Rightarrow B : \mathrm{Ty}$

We derive from this the types Ctx, Var, and Ren, specialised to the notion of Ty

We define the indexed inductive type $\mathrm{Tm} : \mathrm{Ctx} \to \mathrm{Ty} \to \mathrm{Type}$ to be the inductive family generated by:

- For every $v : \mathrm{Var}\ \Gamma\ A$, a *variable*, $\mathrm{V}\ v : \mathrm{Tm}\ \Gamma\ A$
- For every $t : \mathrm{Tm}\ (\Gamma \vdash A)\ B$, an *abstraction*, $\mathrm{Lam}\ t : \mathrm{Tm}\ \Gamma\ (A \Rightarrow B)$
- For every $t : \mathrm{Tm}\ \Gamma\ (A \Rightarrow B)$ and $s : \mathrm{Tm}\ \Gamma\ A$, an *application*, $\mathrm{App}\ t\ s : \mathrm{Tm}\ \Gamma\ B$

STLC Syntax in Agda

In Agda, we can write the above defenitions as follows:

```
data Ty : Type where
  Base : Ty
  _⇒_ : Ty → Ty → Ty

Ctx = Ctx Ty
Var = Var Ty
Ren = Ren Ty

data Tm : Ctx → Ty → Type where
  V : {Γ : Ctx} {A : Ty} (v : Var Γ A) → Tm Γ A
  Lam : {Γ : Ctx} {A B : Ty} (t : Tm (Γ ∔ A) B) → Tm Γ (A ⇒ B)
  App : {Γ : Ctx} {A B : Ty} (t : Tm Γ (A ⇒ B)) (s : Tm Γ A) → Tm Γ B
```

Note that we have neither defined substitution nor included any reduction laws, so this on its own is not a CCC Category

Evaluation

Let's imagine that the above yields a CCC Category $\sigma\iota\nu$

In constructing a CCC preserving functor $\sigma\iota\nu \to \mathscr{C}$, the only degree of freedom is the X : ty to which we send Base:

$$
\begin{aligned}
&[\![\ A\ ]\!] && : \mathsf{ty} \\
&[\![\ \mathsf{Base}\ ]\!] && = X \\
&[\![\ A \Rightarrow B\ ]\!] && = [\![\ A\ ]\!] \Rrightarrow [\![\ B\ ]\!]
\end{aligned}
$$

$$
\begin{aligned}
&[\![\ t\ ]\!] && : \mathsf{tm}\ [\![\ \Gamma\ ]\!]\ [\![\ A\ ]\!] \\
&[\![\ \mathsf{V}\ v\ ]\!] && = \mathsf{makeVar}\ v \\
&[\![\ \mathsf{Lam}\ t\ ]\!] && = \Lambda\ [\![\ t\ ]\!] \\
&[\![\ \mathsf{App}\ t\ s\ ]\!] && = app\ [\![\ t\ ]\!]\ [\![\ s\ ]\!]
\end{aligned}
$$

Sets

Set is Cartesian Closed, and can be given a contextual structure

The eliminator into Set is:

```
X : Type
X = {!!}

el : Ty → Type
el Base = X
el (A ⇒ B) = el A → el B

els : Ctx → Type
els Γ = Els el Γ

Eval : {Γ : Ctx} {A : Ty} → Tm Γ A → els Γ → el A
Eval (V v) es = derive es v
Eval (Lam t) es = λ e → (Eval t) (es ⊕ e)
Eval (App t s) es = (Eval t es) (Eval s es)
```

Church Numerals

In order to test our eliminator, we define some Church arithmetic:

```
ChurchType : Ty → Ty
ChurchType A = (A ⇒ A) ⇒ A ⇒ A

Two : {Γ : Ctx} {A : Ty} → Tm Γ (ChurchType A)
Two = Lam (Lam (App (V (sv zv)) (App (V (sv zv)) (V zv))))

Plus : {Γ : Ctx} {A : Ty} → Tm Γ (ChurchType A ⇒ ChurchType A ⇒ ChurchType A)
Plus = Lam (Lam (Lam (Lam (App (App (V (sv (sv (sv zv)))) (V (sv zv)))
                               (App (App (V (sv (sv zv))) (V (sv zv))) (V zv))))))
```

Two Plus Two

We form the expression representing $2 + 2$ and evaluate it:

```
sum : Tm ∅ (ChurchType Base)
sum = App (App Plus Two) Two

semantic = Eval sum
```

Typing '`C-c C-d semantic RET`' into emacs yields that semantic has type els $\emptyset \to (X \to X) \to X \to X$

Similarly, '`C-c C-n semantic RET`' yields that semantic normalises to $\lambda\ es\ s\ z \to s\ (s\ (s\ (s\ z)))$

The function representing this expression has the form of a Church numeral
We just need some way of extracting a normal form form this!

Normals and Neutrals

The following mutually inductive types allow us to express precisely the β-reduced and η-long normal forms:

We define the indexed inductive type $\mathrm{Ne} : \mathrm{Ctx} \to \mathrm{Ty} \to \mathrm{Type}$ to be the inductive family generated by:

- For every $v : \mathrm{Var}\ \Gamma\ A$, a variable, $\mathrm{VN}\ v : \mathrm{Ne}\ \Gamma\ A$
- For every $M : \mathrm{Ne}\ \Gamma\ (A \Rightarrow B)$ and $N : \mathrm{Nf}\ \Gamma\ A$, an application, $\mathrm{APP}\ M\ N : \mathrm{Ne}\ \Gamma\ B$

We define the indexed inductive type $\mathrm{Nf} : \mathrm{Ctx} \to \mathrm{Ty} \to \mathrm{Type}$ to be the inductive family generated by:

- For every $M : \mathrm{Ne}\ \Gamma\ \mathrm{Base}$, an embedded neutral $\mathrm{NEU}\ M : \mathrm{Nf}\ \Gamma\ \mathrm{Base}$
- For every $N : \mathrm{Nf}\ (\Gamma \dotplus A)\ B$, an abstraction, $\mathrm{LAM}\ N : \mathrm{Nf}\ \Gamma\ (A \Rightarrow B)$

Normals and Neutrals in Agda

In Agda we have:

```
data Nf : Ctx → Ty → Type
data Ne : Ctx → Ty → Type

data Ne where
  VN : {Γ : Ctx} {A : Ty} → Var Γ A → Ne Γ A
  APP : {Γ : Ctx} {A B : Ty} → Ne Γ (A ⇒ B) → Nf Γ A → Ne Γ B

data Nf where
  NEU : {Γ : Ctx} → Ne Γ Base → Nf Γ Base
  LAM : {Γ : Ctx} {A B : Ty} → Nf (Γ ∔ A) B → Nf Γ (A ⇒ B)
```

A First Attempt at Normalisation

All that we know at the moment is how to eliminate into Set, so let's take what we know and go as far as possible

Now, normal forms of type A are not a set because we've left the context unspecified; let's wishfully put all of the contexts together and ignore their presence when forming applications

Thus, if A is a type, then Nf A is a set, and for M : Ne ($A \Rightarrow B$) and N : Nf A, the application APP M N always makes sense

We are going to choose to set X = Nf Base. The elements of $[\![\ A \ ]\!]$ are then known as the *semantic elements* of type A

Quote and Unquote

Given a term $t : \mathsf{Tm}\ \Gamma\ A$, we have that $[\![\ t\ ]\!] : [\![\ \Gamma\ ]\!] \to [\![\ A\ ]\!]$

In order to obtain a normal form, we need a recipes for:
a] turning a semantic element into a normal form (*quote*)
b] cooking up semantic elements (*unquote*)

We will see that the natural domain for defining unquote is on neutral terms. We thus want to define (by mutual induction) the following functions for each type A:

$$\mathsf{q}_A : [\![\ A\ ]\!] \to \mathsf{Nf}\ A$$
$$\mathsf{u}_A : \mathsf{Ne}\ A \to [\![\ A\ ]\!]$$

Unquote

The definition of unquote is the more direct one

At Base, we just need to turn a neutrals into a normal

At $A \Rightarrow B$, we have a neutral $M : \mathsf{Ne}\ (A \Rightarrow B)$ and a semantic element $s : [\![\ A\ ]\!]$
We can turn that semantic element into a normal form using quote, form the application, and then unquote in order to obtain a semantic element

$$\mathsf{u}_A\ (M : \mathsf{Ne\ A}) : [\![\ A\ ]\!]$$
$$\mathsf{u}_{\mathsf{Base}}\ (M : \mathsf{Ne\ Base}) = \mathsf{NEU}\ M$$
$$\mathsf{u}_{A \Rightarrow B}\ (M : \mathsf{Ne}\ (A \Rightarrow B)) = (s : [\![\ A\ ]\!]) \mapsto \mathsf{u}_B\ \left(\mathsf{APP}\ M\ \left(\mathsf{q}_A\ s\right)\right)$$

Note that forming the application required M being a neutral term

Quote

Quote is a bit more subtle

At Base, the semantic elements are already normal forms

At $A \Rightarrow B$, we have a function $f : [\![A]\!] \to [\![B]\!]$
We can quote a semantic element of type B, and then apply LAM to get a normal form of type $A \Rightarrow B$
All that we need now is a semantic element of type A; we obtain this by unquoting a neutral variable

$$\mathsf{q}_A\ (s : [\![A]\!]) : \mathsf{Nf}\ A$$
$$\mathsf{q}_{\mathsf{Base}}\ (N : \mathsf{Nf\ Base}) = N$$
$$\mathsf{q}_{A \Rightarrow B}\ (f : [\![A]\!] \to [\![B]\!]) = \mathsf{LAM}\ (\mathsf{q}_B\ (f\ (\mathsf{u}_A\ (\mathsf{VN}\ zv))))$$

Putting it Together

We have:

$$\llbracket\ t\ \rrbracket : \llbracket\ \Gamma\ \rrbracket \to \llbracket\ A\ \rrbracket$$
$$\mathsf{q}_A : \llbracket\ A\ \rrbracket \to \mathsf{Nf}\ A$$
$$\mathsf{u}_A : \mathsf{Ne}\ A \to \llbracket\ A\ \rrbracket$$

In order to cook up the neutral forms to unquote into a indexed list of semantic elements in $\llbracket\ \Gamma\ \rrbracket$, we use the same trick as before and use neutral variables Instead of making them all zero variables, though, we consider the identity neutral substitution $\mathsf{idNes}\ \Gamma : \mathsf{Nes}\ \Gamma\ \Gamma$. We then have:

$$\mathsf{q}_A\ \left(\llbracket\ t\ \rrbracket\ \left(\mathsf{us}_\Gamma\ (\mathsf{idNe}\ \Gamma)\right)\right) : \mathsf{Nf}\ A$$

Accounting for Contexts

By adding in contexts, we are no longer evaluating in Set

At the base type, we have $X = \mathsf{Nf}\ \mathsf{Base} : \mathsf{Ctx} \to \mathsf{Type}$

So the terms of our new theory are *'context indexed set families'*

We thus have that $[\![\ A\ ]\!]_\Gamma$ is a set for every context Γ

At the base type $[\![\ \mathsf{Base}\ ]\!]_\Gamma = \mathsf{Nf}\ \Gamma\ \mathsf{Base}$

At arrow types, we naively set $[\![\ A \Rightarrow B\ ]\!]_\Gamma = [\![\ A\ ]\!]_\Gamma \to [\![\ B\ ]\!]_\Gamma$

Quote and unquote get refined to functions:

$$\mathsf{q}_{A,\Gamma} : [\![\ A\ ]\!]_\Gamma \to \mathsf{Nf}\ \Gamma\ A$$
$$\mathsf{u}_{A,\Gamma} : \mathsf{Ne}\ \Gamma\ A \to [\![\ A\ ]\!]_\Gamma$$

Changing Contexts

The most obvious sleight of hand in our previous formulation came when we used a zero variable to produce a neutral term of type A in the definition of quote

Suppose that we are defining $\mathsf{q}_{A \Rightarrow B\ ,\ \Gamma}\ f$

Then f is a function $f : [\![\ A\ ]\!]_\Gamma \to [\![\ B\ ]\!]_\Gamma$

Meanwhile zv naturally has type $\mathsf{Var}\ (\Gamma \dot{+} A)\ A$, thus

$$\mathsf{q}_{A\ ,\ \Gamma \dot{+} A}\ (\mathsf{VN}\ zv) : [\![\ A\ ]\!]_{\Gamma \dot{+} A}$$

So we have a problem, because we need to evaluate f on a semantic element in a more general context

Changing Contexts *[cont.]*

To see how to resolve this, suppose that f arises from unquoting

We previously had:

$$\mathsf{u}_{A \Rightarrow B\,,\,\Gamma}\ M\ s = \mathsf{u}_{B\,,\,\Gamma}\ \left(\mathsf{APP}\ M\ \left(\mathsf{q}_{A\,,\,\Gamma}\ s\right)\right)$$

We are expecting $s : [\![\ A\ ]\!]_{\Gamma}$, but what if instead we get $s : [\![\ A\ ]\!]_{\Gamma \dotplus A}$?

In order to form the application, we need to weaken M to context $\Gamma \dotplus A$

This can be done if we are given a renaming of type $\mathsf{Ren}\ (\Gamma \dotplus A)\ \Gamma$

Renaming Normals and Neutrals

We define the contravariant action of renamings on normals and neutrals

We just propagate into subexpressions until we hit a variable

Variables can be replaced with variables (or more generally neutrals), but not with arbitrary terms

$$\_[\_]\mathsf{Ne} : \{\Delta\ \Gamma : \mathsf{Ctx}\}\ \{A : \mathsf{Ty}\} \to \mathsf{Ne}\ \Gamma\ A \to \mathsf{Ren}\ \Delta\ \Gamma \to \mathsf{Ne}\ \Delta\ A$$
$$\_[\_]\mathsf{Nf} : \{\Delta\ \Gamma : \mathsf{Ctx}\}\ \{A : \mathsf{Ty}\} \to \mathsf{Nf}\ \Gamma\ A \to \mathsf{Ren}\ \Delta\ \Gamma \to \mathsf{Nf}\ \Delta\ A$$

$$\mathsf{VN}\ v\ [\ \sigma\ ]\mathsf{Ne} = \mathsf{VN}\ (\mathsf{derive}\ \sigma\ v)$$
$$\mathsf{APP}\ M\ N\ [\ \sigma\ ]\mathsf{Ne} = \mathsf{APP}\ (M\ [\ \sigma\ ]\mathsf{Ne})\ (N\ [\ \sigma\ ]\mathsf{Nf})$$

$$\mathsf{NEU}\ M\ [\ \sigma\ ]\mathsf{Nf} = \mathsf{NEU}\ (M\ [\ \sigma\ ]\mathsf{Ne})$$
$$\mathsf{LAM}\ \{A = A\}\ N\ [\ \sigma\ ]\mathsf{Nf} = \mathsf{LAM}\ (N\ [\ \mathsf{W}_2 Ren\ A\ \sigma\ ]\mathsf{Nf})$$

Correcting the Exponential

With our notion of the action of renamings, suppose we are given:
$M : \mathsf{Ne}\ \Gamma\ (A \Rightarrow B)$, $\sigma : \mathsf{Ren}\ \Delta\ \Gamma$, and $s : [\![\ A\ ]\!]_\Delta$, then:

$$\mathsf{u}_{B,\ \Delta}\ \big(\mathsf{APP}\ (M\ [\ \sigma\ ]\mathsf{Ne})\ \big(\mathsf{q}_{A,\ \Delta}\ s\big)\big) : [\![\ B\ ]\!]_\Delta$$

This suggests that we need to update our definition of $[\![\ A \Rightarrow B\ ]\!]_\Gamma$

$$\begin{aligned}
&\mathsf{El} : \mathsf{Ctx} \to \mathsf{Ty} \to \mathsf{Type} \\
&\mathsf{El}\ \Gamma\ \mathsf{Base} = \mathsf{Nf}\ \Gamma\ \mathsf{Base} \\
&\mathsf{El}\ \Gamma\ (A \Rightarrow B) = \{\Delta : \mathsf{Ctx}\} \to \mathsf{Ren}\ \Delta\ \Gamma \to \mathsf{El}\ \Delta\ A \to \mathsf{El}\ \Delta\ B
\end{aligned}$$

We have thus strengthened the power of semantic elements to be defined on semantic elements in any context Δ, so long as Δ is related to Γ by a renaming

Quote and Unquote

With this we get new definitions of quote and unquote:

$$\mathsf{q} : \{\Gamma : \mathsf{Ctx}\}\ \{A : \mathsf{Ty}\} \to \mathsf{El}\ \Gamma\ A \to \mathsf{Nf}\ \Gamma\ A$$
$$\mathsf{u} : \{\Gamma : \mathsf{Ctx}\}\ \{A : \mathsf{Ty}\} \to \mathsf{Ne}\ \Gamma\ A \to \mathsf{El}\ \Gamma\ A$$

$$\mathsf{q}\ \{A = \mathsf{Base}\}\ N = N$$
$$\mathsf{q}\ \{A = A \Rightarrow B\}\ f = \mathsf{LAM}\ (\mathsf{q}\ (f\ \pi Ren\ (\mathsf{u}\ (\mathsf{VN}\ zv))))$$

$$\mathsf{u}\ \{A = \mathsf{Base}\}\ M = \mathsf{NEU}\ M$$
$$\mathsf{u}\ \{A = A \Rightarrow B\}\ M\ \sigma\ s = \mathsf{u}\ (\mathsf{APP}\ (M\ [\ \sigma\ ]\mathsf{Ne})\ (\mathsf{q}\ s))$$

Evaluation

A term t : Tm Γ A is supposed to evaluate to a term between $[\![\ \Gamma\ ]\!]$ and $[\![\ A\ ]\!]$ in the contextual category of *context indexed sets*

In order to make sense of this, we define:

```
Els : Ctx → Ctx → Type
Els Δ Γ = Tms El Δ Γ
```

And we say that $[\![\ t\ ]\!]$ is a Δ-indexed family of morphisms Els Δ Γ → El Δ A

The Action of Renamings on Semantic Elements

When defining eval, we encounter the case of $[\![\ \mathsf{Lam}\ t\ ]\!]$

As input we take ss : Els $\Delta\ \Gamma$ and need to produce something of type El $\Delta\ (A \Rightarrow B)$

In order to define that, we take additional inputs σ : Ren $\Sigma\ \Delta$ and s : El $\Sigma\ A$

$[\![\ t\ ]\!]$ can take something of type Els $\Sigma\ (\Gamma \div A)$, but in order to have this we need to weaken ss:

$$\_[\_]\mathsf{El} : \{\Delta\ \Gamma : \mathsf{Ctx}\}\ \{A : \mathsf{Ty}\} \to \mathsf{El}\ \Gamma\ A \to \mathsf{Ren}\ \Delta\ \Gamma \to \mathsf{El}\ \Delta\ A$$
$$\_[\_]\mathsf{El}\ \{A = \mathsf{Base}\}\ N\ \sigma = N\ [\ \sigma\ ]\mathsf{Nf}$$
$$\_[\_]\mathsf{El}\ \{A = A \Rightarrow B\}\ f\ \sigma\ \tau\ s = f\ (\sigma \circ_{Ren} \tau)\ s$$

The Definition of Evaluation

We can now define eval:

```
eval : {Γ Δ : Ctx} {A : Ty} → Tm Δ A → Els Γ Δ → El Γ A
eval (V v) ss = derive ss v
eval (Lam t) ss σ s = eval t (mapEls _[ σ ]El ss ⊕ s)
eval {Γ} (App t s) ss = eval t ss (idRen Γ) (eval s ss)
```

As before, we have all of the ingredients necessary to normalise:

```
norm : {Γ : Ctx} {A : Ty} → Tm Γ A → Nf Γ A
norm {Γ} t = q (eval t (mapEls (u ∘ VN) (idRen Γ)))
```

That took only 26 lines of code!

Testing This Out

First, we give an embedding of normal forms into syntax:

$$\imath\mathsf{Ne} : \{\Gamma : \mathsf{Ctx}\}\ \{A : \mathsf{Ty}\} \to \mathsf{Ne}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A$$
$$\imath\mathsf{Nf} : \{\Gamma : \mathsf{Ctx}\}\ \{A : \mathsf{Ty}\} \to \mathsf{Nf}\ \Gamma\ A \to \mathsf{Tm}\ \Gamma\ A$$

$$\imath\mathsf{Ne}\ (\mathsf{VN}\ v) = \mathsf{V}\ v$$
$$\imath\mathsf{Ne}\ (\mathsf{APP}\ M\ N) = \mathsf{App}\ (\imath\mathsf{Ne}\ M)\ (\imath\mathsf{Nf}\ N)$$

$$\imath\mathsf{Nf}\ (\mathsf{NEU}\ M) = \imath\mathsf{Ne}\ M$$
$$\imath\mathsf{Nf}\ (\mathsf{LAM}\ N) = \mathsf{Lam}\ (\imath\mathsf{Nf}\ N)$$

We now consider the example of $2 + 2$ from before:

$$2{+}2{=}4 : \imath\mathsf{Nf}\ (\mathsf{norm}\ \mathsf{sum}) \equiv \mathsf{Lam}\ (\mathsf{Lam}\ (\mathsf{App}\ (\mathsf{V}\ (sv\ zv))\ (\mathsf{App}\ (\mathsf{V}\ (sv\ zv))\ (\mathsf{App}\ (\mathsf{V}\ (sv\ zv))\ (\mathsf{App}\ (\mathsf{V}\ (sv\ zv))\ (\mathsf{V}\ zv))))))$$
$$2{+}2{=}4 = \mathsf{refl}$$