Quotes in λ-calculus and type theroy

Håkon Robbestad Gylterud

Stockholm 2019

Introduction

Quoting in natural language

The password is long.
The password is "long".

Church's Thesis

"Every function is computable."

- Which notion of computable?
- Which functions?

Church's Thesis

"For every function $\mathbb{N} \to \mathbb{N}$ there is a Kleene-index computing the function"

Church's Thesis

"For every function $\mathbb{N} \to \mathbb{N}$ there is a term in λ-calculus computing the function"

- Which encoding?
- What quantifiers?

Church's Thesis

$$\prod_{f:\mathbb{N}\to\mathbb{N}} \sum_{q:\Lambda'(\bot+\top)} \prod_{n:\mathbb{N}} q[r\ n] \rightsquigarrow r\,(f\ n) \tag{1}$$

- $\Lambda'(\bot + \top)$ is the type of terms of the λ'-calculus with one free variable.
- Square brackets are substitution of λ'-terms.
- $r : \mathbb{N} \to \Lambda'\bot$ encodes the numerals as closed λ'-terms.
- `_⇝_ : Λ' X → Λ' X → Set` denotes reduction relation on λ'-terms.

The setting of this talk

We will consider "type theory" in this talk to mean dependent type theory with

- Π-types
- And a limited collections of inductive types: Σ,$+$,$\bot$,$\top$,$\mathtt{Id}$,$\mathbb{N}$,$\mathtt{Fin}$,Λ,Λ',$\rightsquigarrow$
- No η-rules, but we have the ξ-rule (conversion under λ-abstractions)
- No unverses (yet).

Representing (untyped) binding operations in type theory

Binding operations

Ways to express variables and substitution:

- Strings
- De Bruijn indices
- Explicit substitutions
- Combinators

The following representation is due to Bird and Paterson.

Type based de Bruijn indices

```
data Λ (X : Set) : Set where
  var : X → Λ X
  л : Λ (X + T) → Λ X
  app : Λ X → Λ X → Λ X
```

Examples:

- `л (var (right ∗))`
- `л (var (left ∗))`
- `л (л (var (left (right ∗))))`

The translation to de Bruijn indices is simply that `right ∗` corresponds to the index 0 and `left x` is `x + 1`.

Type based de Bruijn indices

```
data Λ (X : Set) : Set where
  var : X → Λ X
  л : Λ (X + ⊤) → Λ X
  app : Λ X → Λ X → Λ X
```

Using this representation it is easy to:

- see that Λ is a monad, with substitution as the Kleisli composition.
- define the reduction relation `_⇝_ : Λ X → Λ X → Set`.

λ'-calculus

λ' calculus

λ'-calculus extends λ-calculus with a new binder:

```
data Λ' (X : Set) : Set where
  var : X → Λ' X
  л : Λ' (X + T) → Λ' X
  app : Λ' X → Λ' X → Λ' X
  _'_ : (n : ℕ) → Λ (X + Fin n) → Λ X
```

Examples

- `[x]'x`
- `[]'x`
- λ`x.[]'x`
- `[]'(`λ`x.x)`
- `[x y]'(x y)`

Examples

- [x]'x, or 1 ' (var (right *))
- []'x, or 0 ' (var (left *))
- λx.[]'x, or л (0 ' (var (left (right *))))
- []'(λx.x), or 0 ' (л (var (right *)))
- [x y]'(x y), or 2 ' (app (var (right *)) ())

Quoting terms in λ-calculus

Detour: Church numerals

- $\texttt{ZERO} := \lambda \texttt{fx.x}$
- $\texttt{SUCC} := \lambda \texttt{n.}\lambda \texttt{fx.f(nx)}$

This can be used to build a function $\texttt{c} : \mathbb{N} \to \Lambda \perp$ in type theory.

Detour: Church numerals

- $\texttt{ZERO} := \lambda\texttt{fx.x}$
- $\texttt{SUCC} := \lambda\texttt{n}.\lambda\texttt{fx.f(nx)}$

This can be used to build a function $\texttt{c} : \mathbb{N} \to \Lambda \perp$ in type theory.

Observe:

- Every Church numeral can be typed: $\texttt{(X} \to \texttt{X)} \to \texttt{(X} \to \texttt{X)}$
- In fact these are all such functions (assuming paramatricity).
- The function $\texttt{iterate} : \mathbb{N} \to \texttt{(X} \to \texttt{X)} \to \texttt{(X} \to \texttt{X)}$ is an instance of the elimination principle for $\mathbb{N}$ in type theory.

Detour: Alternative representation of $\mathbb{N}$ in λ-calculus

From Martin-Löf type theory: Induction principle for $\mathbb{N}$:

$$
\frac{
\begin{array}{ll}
\mathtt{x}:\mathbb{N} & \vdash \mathtt{P\ x\ type} \\
 & \vdash \mathtt{c}_0 \ : \ \mathtt{P\ z} \\
\mathtt{x}:\mathbb{N},\mathtt{y}:\mathtt{P(x)} & \vdash \mathtt{c}_1 \ : \ \mathtt{P\ (s\ n)}
\end{array}
}{
\mathtt{x} : \mathbb{N} \vdash \mathtt{elim}\text{-}\mathbb{N}\ \mathtt{P\ x\ c_0\ c_1 : P\ x}
}\ \mathbb{N}\text{-}\mathtt{ELIM}
$$

Computation rules:

$$
\begin{array}{l}
\vdash \mathtt{elim}\text{-}\mathbb{N}\ \mathtt{P\ z}\qquad\ \ \mathtt{c_0\ c_1} \equiv \mathtt{c_0} \\
\vdash \mathtt{elim}\text{-}\mathbb{N}\ \mathtt{P\ (s\ n)\ c_0\ c_1} \equiv \mathtt{c_1\ n\ (elim}\text{-}\mathbb{N}\ \mathtt{P\ n\ c_0\ c_1)}
\end{array}
$$

Alternative representation of $\mathbb{N}$ in λ-calculus

This inspires the following:

- $\texttt{ZERO} := \lambda \texttt{c}_0 \texttt{c}_1.\ \texttt{c}_0$
- $\texttt{SUCC} := \lambda \texttt{n}.\lambda \texttt{c}_0 \texttt{c}_1.\ \texttt{c}_1\ \texttt{n}\ (\texttt{n}\ \texttt{c}_0\ \texttt{c}_1)$

Which gets the computation rules by β-reduction:

- $\texttt{ZERO}\ \texttt{c}_0\ \texttt{c}_1 \rightsquigarrow \texttt{c}_0$
- $(\texttt{SUCC}\ \texttt{n})\ \texttt{c}_0\ \texttt{c}_1 \rightsquigarrow \texttt{c}_1\ \texttt{n}\ (\texttt{n}\ \texttt{c}_0\ \texttt{c}_1)$

This way of encoding extends to many inductive types.

Representing λ-calculus in λ-calculus

```
data Λ (X : Set) : Set where
  var : X → Λ X
  л : Λ (X + ⊤) → Λ X
  app : Λ X → Λ X → Λ X
```

Which inspires the following the representation of λ-calculus in λ-calculus:

```
VAR = λx.  v l a. v x
LAM = λt.  v l a. l t (t v l a)
APP = λt u.v l a. a t u (t v l a) (u v l a)
```

Representing λ'-calculus in λ(')-calculus

The definition we had of terms in Λ' gives a similar representation

```
data Λ' (X : Set) : Set where
  var : X → Λ' X
  л : Λ' (X + T) → Λ' X
  app : Λ' X → Λ' X → Λ' X
  _'_ : (n : ℕ) → Λ (X + Fin n) → Λ X
```

```
VAR   := λx.     v l a q. v x
LAM   := λt.     v l a q. l t (t v l a q)
APP   := λt u.   v l a q. a t u (t v l a q) (u v l a q)
QUOTE := λ n t. v l a q. q n t (t v l a q)
```

Notice: No quotes are used to represent terms.

The reduction relation in λ'-calculus: variable-quoting

A quote will only be encoding variables which it has bound:

`n ' (var (right x))` $\rightsquigarrow$ `VAR (r x)`

Example: We have `[x]'x` $\rightsquigarrow$ `VAR ZERO` but `[]'x` does not reduce.

The reduction relation in λ'-calculus: λ-quoting

Informally, we want, whenever x does not occur in v:

`v ’ (λx.t) ⇝ LAM (x·v ’ t)`

Formally, we need to do some variable yoga:

`associate : Λ ((X + Fin n) + ⊤) → Λ (X + Fin (succ n))`

And we get:

`n ’ (л t) ⇝ app LAM ((succ n) ’ associate f)`

Example

```
[]'(λx.x) ⇝ LAM ([x] ' x)
          ⇝ LAM (VAR ZERO)
```

The reduction relation in λ'-calculus: app-quoting

A trap!

It would be tempting to have:

`v ' (t u)` $\rightsquigarrow$ `APP (v't) (v'u)`

The reduction relation in λ'-calculus: app-quoting

A trap!

It would be tempting to have:

v ' (t u) $\leadsto$ APP (v't) (v'u)

But, that would break confluence when rewriting under quotes:

We would have both:

[y]'((λx.x)y) $\leadsto$ APP (LAM (VAR ZERO)) (VAR ZERO)

. . . and. . .

[y]'((λx.x)y) $\leadsto$ [y]'y $\leadsto$ VAR ZERO

The reduction relation in λ'-calculus: app-quoting

However, this is safe, whenever the head of `t` is a variable in `v`:

`v ' (t u)` $\rightsquigarrow$ `APP (v't) (v'u)`

The reduction relation in λ'-calculus: app-quoting

However, this is safe, whenever the head of `t` is a variable in `v`:

`v ' (t u)` $\rightsquigarrow$ `APP (v't) (v'u)`

Formally: When `head t = right k` for some `k : Fin n`, we have

`n ' (app t u)` $\rightsquigarrow$ `APP (n't) (n'u)`

The reduction relation in λ'-calculus: '-quoting

Finally, we must also be careful when quoting quotes:

Properties of the λ'-calculus

1. Confluence: Rules were carefully chosen for this.
2. Canonicity: For any *normal* term `t : Λ(⊥+Fin(n))` the closed term `n ' t : Λ ⊥` reduces to a normal (quote-free) λ-term.

Properties of the λ'-calculus

1. Confluence: Rules were carefully chosen for this.
2. Canonicity: For any *normal* term `t : Λ(⊥+Fin(n))` the closed term `n ' t : Λ ⊥` reduces to a normal (quote-free) λ-term.

Proof-sketch of 2: By induction on `t`: we have given rules reducing `X't` for each head normal form `t` could have. Each computation rule applies `'` only to subterms of `t`.

Example

Some quoted terms do not normalise:

`Z = [f]'((`λ`x. f (x x)) (`λ`x. f (x x)))` has the property that

`Z` $\rightsquigarrow$ `(APP (VAR ZERO) Z).`

Quoting as an extension of type theory

Representing terms of type theory in λ-calculus

- For $\mathbb{N}$ we got an encoding in λ-calculus by looking at $\mathbb{N}$-elimination.
- Similarly, we can encode our other inductive types:
  - `PAIR` = λ`a b.`λ`p. p a b` for Σ-types.
  - `REFL` = λ`x.`λ`p.p x` for `Id`-types etc
- λ-abstraction will represented by λ-abstraction.

Consistency of type theory

Consistency of type theory can be proven from:

- Canonicity: If $\vdash$ `a : A` then `a` is canonical.
- Normalisation: Every term can be reduced to a normal form.

Extending type theory with new constants

Given a function $\phi : \mathbb{N} \to \mathbb{N}$ in the meta-theory, how do we extend type theory with it?

Extending type theory with new constants

Given a function $\phi : \mathbb{N} \to \mathbb{N}$ in the meta-theory, how do we extend type theory with it?

- Adding a new constant $\mathtt{f}$, and a rule giving $\vdash \mathtt{f} : \mathbb{N} \to \mathbb{N}$ breaks canonicity.

Extending type theory with new constants

Given a function $\phi : \mathbb{N} \to \mathbb{N}$ in the meta-theory, how do we extend type theory with it?

- Adding a new constant $\mathtt{f}$, and a rule giving $\vdash \mathtt{f} : \mathbb{N} \to \mathbb{N}$ breaks canonicity.

But adding a new constant $\mathtt{x}{:}\mathbb{N} \vdash \mathtt{f(x)} : \mathbb{N}$ does not, if...

Extending type theory with new constants

Given a function $\phi : \mathbb{N} \to \mathbb{N}$ in the meta-theory, how do we extend type theory with it?

- Adding a new constant $\mathtt{f}$, and a rule giving $\vdash \mathtt{f} : \mathbb{N} \to \mathbb{N}$ breaks canonicity.

But adding a new constant $\mathtt{x}{:}\mathbb{N} \vdash \mathtt{f(x)} : \mathbb{N}$ does not, if...

we also add (for each $\mathtt{n} : \mathbb{N}$ in the meta theory) a computation rule:

$\mathtt{cf(N[n])} \equiv \mathtt{N[}\phi\ \mathtt{n]}$

where $\mathtt{N[n]}\mathtt{=}\mathtt{s}^n\mathtt{z}$ is the numeral representation of $\mathtt{n}$ in type theory.

Extending type theory with new constants

How much does type theory know about the new constant `f`?

Extending type theory with new constants

How much does type theory know about the new constant `f`?

- Not very much: If ϕ is, say, monotone, the new type theory does not deduce $\mathtt{x}:\mathbb{N},\mathtt{y}:\mathbb{N}\ ,\ \mathtt{p} : \mathtt{x} \leq \mathtt{y} \vdash \mathtt{f(x)} \leq \mathtt{f(y)}$.

Extending type theory with new constants

How much does type theory know about the new constant `f`?

- Not very much: If ϕ is, say, monotone, the new type theory does not deduce $\mathtt{x}:\mathbb{N},\mathtt{y}:\mathbb{N}\ ,\ \mathtt{p} : \mathtt{x} \leq \mathtt{y} \vdash \mathtt{f(x)} \leq \mathtt{f(y)}$.

But we can add:

$$\mathtt{x}:\mathbb{N},\mathtt{y}:\mathbb{N}\ ,\ \mathtt{p} : \mathtt{x} \leq \mathtt{y} \vdash \mathtt{monf\ x\ y\ p} : \mathtt{f(x)} \leq \mathtt{f(y)}$$

And computation rules, which computes `monf N[n] N[m] p` to a witness for every *n* and *m* (from our meta-theory).

Choices

Quoting could be done in several ways:

1. Quoting into an internal representation of type theory syntax (with quoting extensions).
2. Quoting into λ'-calculus

This approach falls into 2.

The typed quoting binder

We first extend our type theory with the rule:

$$\frac{\Gamma \cdot \Delta \;\vdash\; \mathtt{a} \;:\; \mathtt{A}}{\Gamma \;\vdash\; \mathtt{Q}(\Delta)\mathtt{a} \;:\; \Lambda(\mathtt{Fin}\;\|\Delta\|)}\;\mathtt{QUOTE}$$

Examples

- $\vdash \texttt{Q(x:}\mathbb{N}\texttt{)x : } \Lambda\ (\bot\texttt{+1)}$
- $\texttt{x:}\mathbb{N} \vdash \texttt{Q()x : } \Lambda\ \texttt{0}$
- $\vdash \texttt{(}\lambda\texttt{(x:A)} \rightarrow \texttt{Q()x) : A} \rightarrow \Lambda\ \bot$

Computation rules

Now, using the representation we discussed, we add rules to compute Q-abstractions:

This is straight forward for canonical terms:

$\mathtt{Q}(\Delta)(\mathtt{zero}) \equiv \mathtt{zero}$

$\mathtt{Q}(\Delta)(\mathtt{succ\ n}) \equiv \mathtt{app\ SUCC\ (Q}(\Delta)\ \mathtt{n)}$

$\mathtt{Q}(\Delta)(\lambda(\mathtt{x{:}A})\mathtt{t}) \equiv \text{л}\ (\mathtt{Q}(\Delta,\mathtt{x{:}A})\mathtt{t})$

$(\cdots)$

Computation rules

But for eliminators we must make sure that the head of the principle argument is a variable bound by the quote:

$$\mathtt{Q}(\Delta)(\mathtt{elim}\text{-}\mathbb{N}\ \mathtt{P}\ \mathtt{n}\ \mathtt{c}_0\ \mathtt{c}_1)$$
$$\equiv \mathtt{app}\ (\mathtt{Q}(\Delta)\mathtt{n})\ (\mathtt{Q}(\Delta)\mathtt{c}_0)\ (\mathtt{Q}(\Delta,\mathtt{x}:\mathbb{N},\mathtt{p}:\mathtt{P}(\mathtt{x}))\mathtt{c}_1)$$

is only added when the head of $\mathtt{n}$ is a variable in Δ.

Computation rules

But for eliminators we must make sure that the head of the principle argument is a variable bound by the quote:

$$\texttt{Q}(\Delta)(\texttt{elim-}\mathbb{N}\ \texttt{P}\ \texttt{n}\ \texttt{c}_0\ \texttt{c}_1) \\ \quad \equiv \texttt{app}\ (\texttt{Q}(\Delta)\texttt{n})\ (\texttt{Q}(\Delta)\texttt{c}_0)\ (\texttt{Q}(\Delta,\texttt{x}:\mathbb{N},\texttt{p}:\texttt{P}(\texttt{x}))\texttt{c}_1)$$

is only added when the head of `n` is a variable in Δ.

Once we have added rules for each term former, it is straight-forward to show that if $\vdash$ `a : A` and `a` is normal in the extended theory it reduces to a term of canonical form.

Church's Thesis

$$\prod_{f:\mathbb{N}\to\mathbb{N}} \sum_{q:\Lambda'(\bot+\top)} \prod_{n:\mathbb{N}} q[r\ n] \rightsquigarrow r\,(f\ n) \tag{2}$$

The quote operation provides a candidate q, given `f : ℕ → ℕ` namely `Q(x:ℕ)(f x)`.

- Further extensions needed to show the rest of the statement.

Substitution rule

Here is such a further substitution rule:

$$\frac{\Delta,\mathtt{x{:}A} \vdash \mathtt{t(x)} : \mathtt{B(x)} \qquad \Delta \vdash \mathtt{a{:}A}}{\mathtt{(Q(}\Delta\mathtt{,x{:}A)t(x))}\ \mathtt{[Q(}\Delta\mathtt{)a]} \rightsquigarrow \mathtt{Q(}\Delta\mathtt{)t(a)}}\ \text{Q-SUBST}$$

Church thesis from Q-SUBST (proof sketch)

By induction one can show that `r n = Q()n`.

So given $\texttt{f} : \mathbb{N} \to \mathbb{N}$, we let $\texttt{q} := \texttt{Q(x:}\mathbb{N}\texttt{)(f x)}$, and must prove $\texttt{q [r n]} \rightsquigarrow \texttt{r (f n)}$:

$$
\begin{aligned}
\texttt{q [r n]} &= \texttt{q [Q()n]} \\
&\equiv \texttt{Q(x:}\mathbb{N}\texttt{)(f x)[Q()n]} \\
&\rightsquigarrow \texttt{Q()(f n)} \\
&= \texttt{r (f n)}
\end{aligned}
$$

The reduction step uses `Q-SUBST`.

Current status

- Definition of λ'-calculus and substitution formalised in Agda.
- Still many proofs to formalise (confluence, normalisation)
- An interpreter implemented in Haskell (and a small programming language based on the calculus).
- Ongoing: Giving the computation rules for `Q-SUBST` and proving normalisation and canonicity for these.