Quoting operations as extensions of λ-calculus and type theory

Håkon Robbestad Gylterud

Tuesday February 26, 10:15–12:00

Abstract

Going back to Gödel, we know that many formal languages have the ability to represent its own syntax. The operation which turns an expression into its internal representation is called quoting. For programming languages, one can also ask if they can internally represent their own evaluation function. Work by Brown and Palsberg[0] show that this is even possible to some extent for strongly normalising languages.

Quoting usually a meta-theoretical operation. However, some programming languages, such as LISP or Scheme, have this as internal operation in the language. In this talk I will present extensions of λ-calculus and type theory with internal quoting operations. They differ from the LIPS or Scheme equivalents by being confluent while allowing reductions under the quote.

Quoting in natural language

Quoting in natural language

The password is long.

The password is "long".

Quoting in Scheme

Quoting in Scheme: Example 0

```
 > (' (lambda (x) x))
(lambda (x) x)
```

Quoting in Scheme: Example 0

```
> (' (lambda (x) x))
(list 'lambda (list 'x) 'x)
```

Quoting in Scheme: Example 1

Reductions cannot be de done under the quote:

```
> (' ((lambda (x) x) 'z)
(list ('lambda (list 'x) 'x) 3)
```

which is very different from the quote of 3, (which is the number 3 itself).

Quoting in Scheme: Example 2

The following expression gives an error:

```
> (((lambda (y) (lambda (x) y)) (eval 'x)) 3)
Error: eval: unbound variable: x
```

Quoting in Scheme: Example 2

The following expression gives an error:

```
> (((lambda (y) (lambda (x) y)) (eval 'x)) 3)
Error: eval: unbound variable: x
```

but the following β-equivalent expression returns a value:

```
> ((lambda (x) (eval 'x)) 3)
3
```

Representing λ-calculus in λ-calculus

Church numerals

The church numeral `c[n]`, represents the natural number `n`, by a term in λ-calculus:

- `c[z]` = λ`f x. x`
- `c[s n]` = λ`f x. f c[n]`

This could be typed as: `c[n] : Π x → (x → x) → (x → x)`.

Moral: Applying a Church numeral, corresponds to a kind of recursor.

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

From MLTT: Induction principle for $\mathbb{N}$:

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

Computation rules:

$$\vdash \texttt{elim-}\mathbb{N}\ \texttt{P z}\ \ \texttt{c}_0\ \texttt{c}_1 \equiv \texttt{c}_0$$
$$\vdash \texttt{elim-}\mathbb{N}\ \texttt{P (s n)}\ \texttt{c}_0\ \texttt{c}_1 \equiv \texttt{c}_1\ \texttt{n (elim-}\mathbb{N}\ \texttt{P n c}_0\ \texttt{c}_1\texttt{)}$$

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

Given a natural number `n` we can define an alternative representation `r[n]`, inspired by the elimination rule for $\mathbb{N}$:

- `r[z]` = λ c_0 c_1. c_0
- `r[s n]` = λ c_0 c_1. c_1 `(r[n])` `(r[n]` c_0 c_1`)`

Representing λ-calculus in type theory

```
data Λ (X : Set) : Set where
  var : X → Λ X
  л : Λ (X + 1) → Λ X
  app : Λ X → Λ X → Λ X
```

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

```
var = λx    cv cl ca. cv x
л   = λt    cv cl ca. cl t (t cv cl ca)
app = λt u cv cl ca. ca t u (t cv cl ca) (u cv cl ca)
```

Extending λ-calculus with a quote operator

Quoting as a binder

Extend the syntax of λ-calculus with a new binder:

```
t term    X list of distinct variables
```

```
 X't term
```

```
FV (X't) = FV t \ X
```

Examples

- [x]'x
- []'x
- λx.[]'x
- []'(λx.x)
- [x]'(x y)
- [y]'(x y)

Representing λ'-calculus in type theory

```
data Λ (X : Set) : Set where
  var : X → Λ X
  л : Λ (X + 1) → Λ X
  app : Λ X → Λ X → Λ X
  _'_ : (n : ℕ) → Λ (X + Fin n) → Λ X
```

Using this representation, we can for instance implement a substitution operation `subst : Λ(X+1) → Λ(X) → Λ(X)`

Rewriting under quotes

We want to be able to rewrite under the quote – i.e.:

$$\texttt{t} \rightsquigarrow \texttt{u} \quad \Rightarrow \quad \texttt{X't} \rightsquigarrow \texttt{X'u}$$

Computation rules for the quote

Computation rules for the quote: λ

The case for λ-abstraction:

$$\texttt{X'}(\lambda \texttt{y.t}) \rightsquigarrow (\texttt{Л}\ (\texttt{X.y}\ '\ \texttt{t}))$$

. . . assuming $\texttt{x}$ is not in $\texttt{X}$.

Computation rules for the quote: variables

`X'(Xi)` $\rightsquigarrow$ `(var (r[i]))`

Example: `[x,y]'y` $\rightsquigarrow$ `var (r[s z])`.

Computation rules for the quote: Application

It would be tempting to have:

`X'(t u)` $\rightsquigarrow$ `(app (X't) (X'u))`

But, that would break confluence when rewriting under quotes.

Computation rules for the quote: Application

However, this is safe:

$\texttt{X'(x u)} \rightsquigarrow \texttt{(app (X'x) (X'u))}$

when $\texttt{x} \equiv \texttt{X i}$ for some $\texttt{i}$.

Computation rules for the quote: Application

In fact, we can have

`X'(t u)` $\rightsquigarrow$ `(app (X't) (X'u))`

whenever the head of `t` is a variable in `X`.

Computation rules for the quote: Quote

Finally, we need rules for quoting quotes: needing first a representation of the quote constructor in λ-calculus:

```
var = λx    cv cl ca cq. cv x
л   = λt    cv cl ca cq. cl t (t cv cl ca cq)
app = λt u cv cl ca cq. ca t u (t cv cl ca cq) (u cv cl ca
quote = λ n t cv cl ca cq. cq n t (t cv cl ca cq)
```

Computation rules for the quote: Quote

Again, we cannot always have:

$$\texttt{X'(Y'u)} \rightsquigarrow \texttt{quote (r[}\|\texttt{Y}\|\texttt{]) (X.Y'u)}$$

But must require that the head of `u` is a variable in X, and X and Y must be disjoint.

Example computations

- `[x]'x` $\rightsquigarrow$ `(var (r[0]))`
- `[]'x` is normal (`x` is free).
- λ`x.[]'x` is normal.
- `[]'(`λ`x.x)` $\rightsquigarrow$ `(`Λ `(var (r[0])))`
- `[x]'(x y)` $\rightsquigarrow$ `(app (var (r[0])) y)`
- `[y]'(x y)` is normal.

Observation

If `t` is normal and `FV(t)` are all in `X`, then `X't` reduces to a `'`-free term.

Proof-sketch: 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]'((λx. f (x x)) (λx. f (x x)))` has the property that

`Z` $\rightsquigarrow$ `(app (var (r[0])) Z)`.

Quoting as an extension of MLTT

Consistency of type theory

Consistency of MLTT can be proven from:

- Canonicity: Every normal $\vdash \mathtt{a} : \mathtt{A}$ 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 can we extend type theory with it?

Extending type theory with new constants

Adding a constant $\vdash \mathsf{c}\phi : \mathbb{N} \to \mathbb{N}$ breaks canonicity.

Extending type theory with new constants

Adding a constant $\vdash \mathsf{c}\phi : \mathbb{N} \to \mathbb{N}$ breaks canonicity.

But adding a constant $\mathtt{x} : \mathbb{N} \vdash \mathsf{c}\phi(\mathtt{x}) : \mathbb{N}$ does not...

Extending type theory with new constants

Adding a constant $\vdash \mathtt{c}\phi : \mathbb{N} \to \mathbb{N}$ breaks canonicity.

But adding a constant $\mathtt{x} : \mathbb{N} \vdash \mathtt{c}\phi(\mathtt{x}) : \mathbb{N}$ does not...

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

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

where $\mathtt{N[n]}=\mathtt{s}^n\mathtt{z}$ 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 $\mathtt{c}\phi$?

- Not very much: If ϕ is, say, monotone, type theory does not know it.

But we can add:

$$\mathtt{x}{:}\mathbb{N},\mathtt{y}{:}\mathbb{N}\ ,\ \mathtt{p}\ :\ \mathtt{x}\leq\mathtt{y}\vdash\mathtt{mon}\phi\ \mathtt{p}\ :\ \mathtt{c}\phi(\mathtt{x})\leq\mathtt{c}\phi(\mathtt{y})$$

And computation rules, which computes $\mathtt{mon}\phi\ \ (\mathtt{Nn})(\mathtt{Nm})$.

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

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

Examples

- $\vdash \mathtt{Q(x{:}\mathbb{N})x} : \Lambda\ \mathtt{(0+1)}$
- $\mathtt{x{:}\mathbb{N}} \vdash \mathtt{Q()x} : \Lambda\ \mathtt{0}$
- $\vdash \mathtt{(\lambda(x{:}A) \to Q()x)} : \mathtt{A} \to \Lambda\ \mathtt{0}$

Computation rules

We add computation rules for $\mathtt{Q}$ similar to those we had in the λ'-calculus. As an example, here are the computations rules for quoting natural numbers:

$$\mathtt{Q(\Delta)z} \equiv \mathtt{л\ л\ v_+}$$

$$\mathtt{Q(\Delta)(s\ t)} \equiv \mathtt{л\ л\ (app\ (app\ v\ (Q(\Delta)t)\ (app\ (app\ (Q(\Delta)t)\ v)\ v_+))}$$

$$\mathtt{Q(\Delta)(elim\text{-}\mathbb{N}\ P\ u\ c_0\ c_1)} \equiv \mathtt{app\ (app\ (app\ (Q(\Delta)u)\ (Q(\Delta)c_0))\ (Q(\Delta,x:\mathbb{N},y:P(x))c_1)}$$

The quote rule for the eliminator applies only whenever the head of $\mathtt{u}$ is in Δ.

Here $\mathtt{v}$ and $\mathtt{v_+}$ are deBruijn-indices.

Church's thesis in type theory

Given an internal definition of $\mathtt{r[-]} : \mathbb{N} \to \Lambda\ \mathtt{0}$, we propose the following alternative internalisation of church thesis.

$$\prod(\mathtt{f} : \mathbb{N} \to \mathbb{N})\ \sum(\mathtt{q} : \Lambda\ \mathtt{(0+1)})\ \prod\ \mathtt{(n:}\mathbb{N}\mathtt{)}\ \mathtt{(subst\ q\ (r[n]))} \rightsquigarrow \mathtt{r[f\ n]}$$

(Where $\mathtt{subst} : \Lambda\mathtt{(X+1)} \to \Lambda\mathtt{(X)} \to \Lambda\mathtt{(X)}$ is the internally defined substitution of λ'-terms)

Church's thesis in type theory

Given an internal definition of $\mathtt{r[-]} : \mathbb{N} \to \Lambda\ \mathtt{0}$, we propose the following alternative internalisation of church thesis.

$$\prod(\mathtt{f} : \mathbb{N} \to \mathbb{N})\ \sum(\mathtt{q} : \Lambda\ (\mathtt{0+1}))\ \prod\ (\mathtt{n}:\mathbb{N})\ (\mathtt{subst\ q\ (r[n])}) \rightsquigarrow \mathtt{r[f\ n]}$$

(Where $\mathtt{subst} : \Lambda(\mathtt{X+1}) \to \Lambda(\mathtt{X}) \to \Lambda(\mathtt{X})$ is the internally defined substitution of λ'-terms)

The quote operation provides a candidate $\mathtt{q}$, given $\mathtt{f} : \mathbb{N} \to \mathbb{N}$ namely $\mathtt{Q(x:\mathbb{N})(f\ x)}$.

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

Substitution rule

Here is such a further substitution rule:

$$\frac{\texttt{Δ,x:A} \vdash \texttt{t(x)} : \texttt{B(x)} \quad \texttt{Δ} \vdash \texttt{a:A}}{\texttt{subst (Q(Δ,x:A)t(x)) (Q(Δ)a)} \rightsquigarrow \texttt{Q(Δ)t(a)}}\ \texttt{Q-SUBST}$$

Where `t(a)` denotes the substitution of `x` with `a` on the type theory level, and `subst` is the internally defined

Church thesis from Q-SUBST (proof sketch)

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

So given `f : ℕ → ℕ`, we let `q := Q(x:ℕ)(f x)`, and must prove `(subst q (r[n])) ⇝ r[f n]`:

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

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.