Towards Quotient Inductive Types in Observational Type Theory

Thiago Felicissimo & Nicolas Tabareau

March 12, 2025

Quotients in mathematics

Quotients are ubiquitous in mathematics:

Quotients in mathematics

Quotients are ubiquitous in mathematics:

- Construction of integers, rationals, reals

Quotients in mathematics

Quotients are ubiquitous in mathematics:

- Construction of integers, rationals, reals
- Quotient of a ring by an ideal (e.g., $\mathbb{Z}/n\mathbb{Z}$)

Quotients in mathematics

Quotients are ubiquitous in mathematics:

- Construction of integers, rationals, reals
- Quotient of a ring by an ideal (e.g., $\mathbb{Z}/n\mathbb{Z}$)
- Projective space of a vector space

Quotients in mathematics

Quotients are ubiquitous in mathematics:

- Construction of integers, rationals, reals
- Quotient of a ring by an ideal (e.g., $\mathbb{Z}/n\mathbb{Z}$)
- Projective space of a vector space
- …

Quotients in mathematics

Quotients are ubiquitous in mathematics:

- Construction of integers, rationals, reals
- Quotient of a ring by an ideal (e.g., $\mathbb{Z}/n\mathbb{Z}$)
- Projective space of a vector space
- …

Unfortunately, quotients can only be formed in Rocq in very specific situations, namely when can define a function escaping the quotient (Mathcomp quotients)

The quotient type in type theory

General quotients can be constructed in type theory using the *quotient type*:

$$\frac{A : \mathrm{Type} \qquad R : A \to A \to \mathrm{Prop} \qquad R \text{ equiv. rel.}}{A/R : \mathrm{Type}} \qquad \frac{t : A}{[t] : A/R}$$

The quotient type in type theory

General quotients can be constructed in type theory using the *quotient type*:

$$\frac{A : \mathrm{Type} \qquad R : A \to A \to \mathrm{Prop} \qquad R \text{ equiv. rel.}}{A/R : \mathrm{Type}} \qquad \frac{t : A}{[t] : A/R}$$

We also need the axiom

$$Q_{=} : R\, x\, y \to [x] =_{A/R} [y]$$

for characterizing equality of A/R

The quotient type in type theory

General quotients can be constructed in type theory using the *quotient type*:

$$\frac{A : \mathrm{Type} \qquad R : A \to A \to \mathrm{Prop} \qquad R \text{ equiv. rel.}}{A/R : \mathrm{Type}} \qquad \frac{t : A}{[t] : A/R}$$

We also need the axiom

$$\mathrm{Q}_= : R\;x\;y \to [x] =_{A/R} [y]$$

for characterizing equality of A/R

Problem Blocks computation, the following closed term is stuck

$$\left(\begin{array}{l}\mathsf{match}\ \mathrm{Q}_=\ * : [\mathsf{true}] =_{\mathsf{Bool}/(\lambda xy.\{*\})} [\mathsf{true}]\ \mathsf{with} \\ \quad \mid \mathsf{refl} \to 0\end{array}\right) : \mathsf{Nat}$$

The quotient type in type theory

General quotients can be constructed in type theory using the *quotient type*:

$$\frac{A : \mathrm{Type} \qquad R : A \to A \to \mathrm{Prop} \qquad R \text{ equiv. rel.}}{A/R : \mathrm{Type}} \qquad \frac{t : A}{[t] : A/R}$$

We also need the axiom

$$\mathrm{Q}_= : R\ x\ y \to [x] =_{A/R} [y]$$

for characterizing equality of A/R

Problem Blocks computation, the following closed term is stuck

$$\left(\begin{array}{l} \mathsf{match}\ \mathrm{Q}_=\ * : [\mathsf{true}] =_{\mathsf{Bool}/(\lambda xy.\{*\})} [\mathsf{true}]\ \mathsf{with} \\ \quad \mid \mathsf{refl} \to 0 \end{array} \right) : \mathsf{Nat}$$

The approach taken by Lean, as it does not care for canonicity

Observational Type Theory (OTT) to the rescue

In *Observational Type Theory*, equality is instead eliminating using a *cast* operator:

$$\frac{A, B : \mathrm{Type} \qquad p : A =_{\mathrm{Type}} B \qquad a : A}{\mathrm{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Observational Type Theory (OTT) to the rescue

In *Observational Type Theory*, equality is instead eliminating using a *cast* operator:

$$\frac{A, B : \text{Type} \qquad p : A =_{\text{Type}} B \qquad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property of OTT Computation rules for cast *never look inside eq. proofs*

$$\text{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} \langle t_1, t_2 \rangle \longrightarrow \langle \text{cast}_{p.1}^{A \rightsquigarrow A'} t_1, \text{cast}_{p.2}^{B \rightsquigarrow B'} t_2 \rangle$$

Observational Type Theory (OTT) to the rescue

In *Observational Type Theory*, equality is instead eliminating using a *cast* operator:

$$\frac{A, B : \text{Type} \qquad p : A =_{\text{Type}} B \qquad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property of OTT Computation rules for cast *never look inside eq. proofs*

$$\text{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} \langle t_1, t_2 \rangle \longrightarrow \langle \text{cast}_{p.1}^{A \rightsquigarrow A'} t_1, \text{cast}_{p.2}^{B \rightsquigarrow B'} t_2 \rangle$$

Thus, we can add many desirable principles *without blocking computation* (Pujet and Tabareau 2022):

Observational Type Theory (OTT) to the rescue

In *Observational Type Theory*, equality is instead eliminating using a *cast* operator:

$$\frac{A, B : \text{Type} \qquad p : A =_{\text{Type}} B \qquad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property of OTT Computation rules for cast *never look inside eq. proofs*

$$\text{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} \langle t_1, t_2 \rangle \longrightarrow \langle \text{cast}_{p.1}^{A \rightsquigarrow A'} t_1, \text{cast}_{p.2}^{B \rightsquigarrow B'} t_2 \rangle$$

Thus, we can add many desirable principles *without blocking computation* (Pujet and Tabareau 2022):

- funext: two functions equal iff pointwise equal

Observational Type Theory (OTT) to the rescue

In *Observational Type Theory*, equality is instead eliminating using a *cast* operator:

$$\frac{A, B : \text{Type} \qquad p : A =_{\text{Type}} B \qquad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property of OTT Computation rules for cast *never look inside eq. proofs*

$$\text{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} \langle t_1, t_2 \rangle \longrightarrow \langle \text{cast}_{p.1}^{A \rightsquigarrow A'} t_1, \text{cast}_{p.2}^{B \rightsquigarrow B'} t_2 \rangle$$

Thus, we can add many desirable principles *without blocking computation* (Pujet and Tabareau 2022):

- funext: two functions equal iff pointwise equal
- propext: two propositions equal iff equivalent

Observational Type Theory (OTT) to the rescue

In *Observational Type Theory*, equality is instead eliminating using a *cast* operator:

$$\frac{A, B : \text{Type} \qquad p : A =_{\text{Type}} B \qquad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property of OTT Computation rules for cast *never look inside eq. proofs*

$$\text{cast}_p^{(A\times B)\rightsquigarrow(A'\times B')}\langle t_1, t_2\rangle \longrightarrow \langle \text{cast}_{p.1}^{A\rightsquigarrow A'}\, t_1, \text{cast}_{p.2}^{B\rightsquigarrow B'}\, t_2\rangle$$

Thus, we can add many desirable principles *without blocking computation* (Pujet and Tabareau 2022):

- funext: two functions equal iff pointwise equal
- propext: two propositions equal iff equivalent
- uip: equality is proof-irrelevant (like in usual mathematics)

Observational Type Theory (OTT) to the rescue

In *Observational Type Theory*, equality is instead eliminating using a *cast* operator:

$$\frac{A, B : \mathrm{Type} \qquad p : A =_{\mathrm{Type}} B \qquad a : A}{\mathrm{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property of OTT Computation rules for cast *never look inside eq. proofs*

$$\mathrm{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} \langle t_1, t_2 \rangle \longrightarrow \langle \mathrm{cast}_{p.1}^{A \rightsquigarrow A'} t_1, \mathrm{cast}_{p.2}^{B \rightsquigarrow B'} t_2 \rangle$$

Thus, we can add many desirable principles *without blocking computation* (Pujet and Tabareau 2022):

- funext: two functions equal iff pointwise equal
- propext: two propositions equal iff equivalent
- uip: equality is proof-irrelevant (like in usual mathematics)
- Quotient types!

Quotient Inductive Types (QITs)

We have quotient types, are we done?

Quotient Inductive Types (QITs)

We have quotient types, are we done? No, we also want *Quotient Inductive Types*:

$$\text{Inductive MSet } (A : \text{Type}) : \text{Type} :=$$

$$\mid [\,] : \text{MSet } A \qquad \mid \_ :: \_ \ (x : A)(m : \text{MSet } A) : \text{MSet } A$$

$$\mid \text{MSet}_{=} \ (x\ y : A)(m : \text{MSet } A) : (x :: y :: m) = (y :: x :: m)$$

Quotient Inductive Types (QITs)

We have quotient types, are we done? No, we also want *Quotient Inductive Types*:

$$\text{Inductive MSet } (A : \text{Type}) : \text{Type} :=$$

$$\mid [\,] : \text{MSet } A \qquad \mid \_ :: \_ \ (x : A)(m : \text{MSet } A) : \text{MSet } A$$

$$\mid \text{MSet}_{=} \ (x\ y : A)(m : \text{MSet } A) : (x :: y :: m) = (y :: x :: m)$$

Correspond to initial models of (non-pure) algebraic theories.

Quotient Inductive Types (QITs)

We have quotient types, are we done? No, we also want *Quotient Inductive Types*:

$$\text{Inductive MSet } (A : \text{Type}) : \text{Type} :=$$

$$\mid [\,] : \text{MSet } A \qquad \mid \_ :: \_ \ (x : A)(m : \text{MSet } A) : \text{MSet } A$$

$$\mid \text{MSet}_{=} \ (x\ y : A)(m : \text{MSet } A) : (x :: y :: m) = (y :: x :: m)$$

Correspond to initial models of (non-pure) algebraic theories.

Functions eliminating a QIT must respect equality:

$$\text{Fixpoint sum} : \text{MSet Nat} \longrightarrow \text{Nat} :=$$

$$\text{match } l \text{ with}$$

$$\mid [\,] \longrightarrow 0 \qquad \mid x :: m \longrightarrow x + (\text{sum } m)$$

$$\mid \text{MSet}_{=} \ x\ y\ m \longrightarrow (\dots) : (x + y + \text{sum } m) = (y + x + \text{sum } m)$$

More QITs

Integers, rationals, …

Inductive Int : Type :=

| 0 : Int

| S (x : Int) : Int

| P (x : Int) : Int

| $\mathrm{Int}_=$ (x : Int) : S (P x) = x = P (S x)

More QITs

Integers, rationals, ...

Inductive Int : Type :=

| 0 : Int

| S $(x : \mathsf{Int})$: Int

| P $(x : \mathsf{Int})$: Int

| $\mathsf{Int}_{=}$ $(x : \mathsf{Int}) : \mathsf{S}\ (\mathsf{P}\ x) = x = \mathsf{P}\ (\mathsf{S}\ x)$

(Free) Groups, monoids, rings, ...

Inductive Mon $(A : \mathsf{Type})$: Type :=

| 1 : Mon A | gen $(a : A)$: Mon A

| $\_ \cdot \_$ $(x\ y : \mathsf{Mon}\ A) : \mathsf{Mon}\ A$

| $\mathsf{Mon}^{1}_{=}$ $(x : \mathsf{Mon}\ A) : 1 \cdot x = x = x \cdot 1$

| $\mathsf{Mon}^{\cdot/\mathrm{as}}_{=}$ $(x\ y\ z : \mathsf{Mon}\ A) : x \cdot (y \cdot z) = (x \cdot y) \cdot z$

More QITs

Integers, rationals, ...

Inductive Int : Type :=

| 0 : Int

| S (x : Int) : Int

| P (x : Int) : Int

| $\mathsf{Int}_=$ (x : Int) : S (P x) = x = P (S x)

(Free) Groups, monoids, rings, ...

Inductive Mon (A : Type) : Type :=

| 1 : Mon A | gen (a : A) : Mon A

| _ · _ (x y : Mon A) : Mon A

| $\mathsf{Mon}^{1}_{=}$ (x : Mon A) : $1 \cdot x = x = x \cdot 1$

| $\mathsf{Mon}^{\cdot/\mathrm{as}}_{=}$ (x y z : Mon A) : $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

Syntax of prog. languages

Inductive Tm : Type :=

| S : Tm | K : Tm | _ · _ (x y : Tm) : Tm

| $\mathsf{Tm}^{\mathsf{K}}_{=}$ (x y : Tm) : $\mathsf{K} \cdot x \cdot y = x$

| $\mathsf{Tm}^{\mathsf{S}}_{=}$ (x y z : Tm) : $\mathsf{S} \cdot x \cdot y \cdot z = x \cdot z \cdot (y \cdot z)$

More QITs

Integers, rationals, …

Inductive Int : Type :=

$|\ 0 : \mathsf{Int}$

$|\ \mathsf{S}\ (x : \mathsf{Int}) : \mathsf{Int}$

$|\ \mathsf{P}\ (x : \mathsf{Int}) : \mathsf{Int}$

$|\ \mathsf{Int}_{=}\ (x : \mathsf{Int}) : \mathsf{S}\ (\mathsf{P}\ x) = x = \mathsf{P}\ (\mathsf{S}\ x)$

(Free) Groups, monoids, rings, …

Inductive Mon $(A : \mathsf{Type})$: Type :=

$|\ 1 : \mathsf{Mon}\ A \qquad |\ \mathsf{gen}\ (a : A) : \mathsf{Mon}\ A$

$|\ \_ \cdot \_\ (x\ y : \mathsf{Mon}\ A) : \mathsf{Mon}\ A$

$|\ \mathsf{Mon}^{1}_{=}\ (x : \mathsf{Mon}\ A) : 1 \cdot x = x = x \cdot 1$

$|\ \mathsf{Mon}^{\cdot/\mathsf{as}}_{=}\ (x\ y\ z : \mathsf{Mon}\ A) : x \cdot (y \cdot z) = (x \cdot y) \cdot z$

Syntax of prog. languages

Inductive Tm : Type :=

$|\ \mathsf{S} : \mathsf{Tm} \qquad |\ \mathsf{K} : \mathsf{Tm} \qquad |\ \_ \cdot \_\ (x\ y : \mathsf{Tm}) : \mathsf{Tm}$

$|\ \mathsf{Tm}^{\mathsf{K}}_{=}\ (x\ y : \mathsf{Tm}) : \mathsf{K} \cdot x \cdot y = x$

$|\ \mathsf{Tm}^{\mathsf{S}}_{=}\ (x\ y\ z : \mathsf{Tm}) : \mathsf{S} \cdot x \cdot y \cdot z = x \cdot z \cdot (y \cdot z)$

Inductive Tm : Ty → Type :=

$|\ \mathsf{true} : \mathsf{Tm\ bool} \qquad |\ \mathsf{false} : \mathsf{Tm\ bool}$

$|\ \mathsf{if}\ \{A\}(x : \mathsf{Tm\ bool})(t\ u : \mathsf{Tm}\ A) : \mathsf{Tm}\ A$

$|\ \mathsf{Tm}^{\mathsf{if/true}}_{=}\ \{A\}(t\ u : \mathsf{Tm}\ A) : \mathsf{if\ true}\ t\ u = t$

…

Metatheory of QITs in OTT

How can we know that OTT extended with QITs is well-behaved?

Metatheory of QITs in OTT

How can we know that OTT extended with QITs is well-behaved?

Strategy 1 Extend OTT with inductive scheme for QITs

Metatheory of QITs in OTT

How can we know that OTT extended with QITs is well-behaved?

Strategy 1 Extend OTT with inductive scheme for QITs

Problem Inductive schemes are hard to manipulate formally

No go if we want to formally prove normalization

Metatheory of QITs in OTT

How can we know that OTT extended with QITs is well-behaved?

Strategy 1 Extend OTT with inductive scheme for QITs

Problem Inductive schemes are hard to manipulate formally
No go if we want to formally prove normalization

Strategy 2 Encode QITs using inductive types + quotient type Q,
both of which have already been studied in OTT by Pujet and Tabareau

Metatheory of QITs in OTT

How can we know that OTT extended with QITs is well-behaved?

Strategy 1 Extend OTT with inductive scheme for QITs

Problem Inductive schemes are hard to manipulate formally

No go if we want to formally prove normalization

Strategy 2 Encode QITs using inductive types + quotient type Q, both of which have already been studied in OTT by Pujet and Tabareau

Problem Eliminator of encoded QIT does not compute properly

Moreover, construction does not seem even possible for *infinitary* QITs

Metatheory of QITs in OTT

How can we know that OTT extended with QITs is well-behaved?

Strategy 1 Extend OTT with inductive scheme for QITs

Problem Inductive schemes are hard to manipulate formally
No go if we want to formally prove normalization

Strategy 2 Encode QITs using inductive types + quotient type Q,
both of which have already been studied in OTT by Pujet and Tabareau

Problem Eliminator of encoded QIT does not compute properly
Moreover, construction does not seem even possible for *infinitary* QITs

Our strategy Extend OTT with a single *universal* QIT, capable of encoding all QITs

The plan

We have proposed a universal non-indexed QIT, adapting Fiore *et al.*'s QW types:

`https://github.com/thiagofelicissimo/universal-QITs`

Used to define various examples: multisets, SK calculus, finitely branching trees, …

The plan

We have proposed a universal non-indexed QIT, adapting Fiore *et al.*'s QW types:

`https://github.com/thiagofelicissimo/universal-QITs`

Used to define various examples: multisets, SK calculus, finitely branching trees, …

The next steps of our work are:

The plan

We have proposed a universal non-indexed QIT, adapting Fiore *et al.*'s QW types:

`https://github.com/thiagofelicissimo/universal-QITs`

Used to define various examples: multisets, SK calculus, finitely branching trees, …

The next steps of our work are:

1. Formulate an inductive scheme for non-indexed QITs, then prove that they can all be encoded using our universal QIT

The plan

We have proposed a universal non-indexed QIT, adapting Fiore *et al.*'s QW types:

`https://github.com/thiagofelicissimo/universal-QITs`

Used to define various examples: multisets, SK calculus, finitely branching trees, …

The next steps of our work are:

1. Formulate an inductive scheme for non-indexed QITs, then prove that they can all be encoded using our universal QIT
2. Prove that OTT + universal QIT is normalizing, and so has decidable typing

The plan

We have proposed a universal non-indexed QIT, adapting Fiore *et al.*'s QW types:

`https://github.com/thiagofelicissimo/universal-QITs`

Used to define various examples: multisets, SK calculus, finitely branching trees, …

The next steps of our work are:

1. Formulate an inductive scheme for non-indexed QITs, then prove that they can all be encoded using our universal QIT
2. Prove that OTT + universal QIT is normalizing, and so has decidable typing
3. Prove that OTT + universal QIT is consistent (not a consequence of 2!)

The plan

We have proposed a universal non-indexed QIT, adapting Fiore *et al.*'s QW types:

`https://github.com/thiagofelicissimo/universal-QITs`

Used to define various examples: multisets, SK calculus, finitely branching trees, ...

The next steps of our work are:

1. Formulate an inductive scheme for non-indexed QITs, then prove that they can all be encoded using our universal QIT
2. Prove that OTT + universal QIT is normalizing, and so has decidable typing
3. Prove that OTT + universal QIT is consistent (not a consequence of 2!)

From 2 and 3 we can then deduce canonicity of the theory and of encoded QITs

The plan

We have proposed a universal non-indexed QIT, adapting Fiore *et al.*'s QW types:

`https://github.com/thiagofelicissimo/universal-QITs`

Used to define various examples: multisets, SK calculus, finitely branching trees, . . .

The next steps of our work are:

1. Formulate an inductive scheme for non-indexed QITs, then prove that they can all be encoded using our universal QIT
2. Prove that OTT + universal QIT is normalizing, and so has decidable typing
3. Prove that OTT + universal QIT is consistent (not a consequence of 2!)

From 2 and 3 we can then deduce canonicity of the theory and of encoded QITs

Once finished, move to more complex classes of types: indexed QITs and QIITs

The ultimate goal

Once we know OTT+QITs is well-behaved, we can have Rocq with

1. funext: two functions equal iff pointwise equal
2. propext: two propositions equal iff equivalent
3. uip: equality is proof-irrelevant (like in usual mathematics)
4. (Indexed) Inductive types: Nat, List, Vec,…
5. Quotient types
6. Quotient Inductive Types: MSet, Int, Mon, …

all while preserving canonicity, consistency and decidability of typing

The ultimate goal

Once we know OTT+QITs is well-behaved, we can have Rocq with

1. funext: two functions equal iff pointwise equal
2. propext: two propositions equal iff equivalent
3. uip: equality is proof-irrelevant (like in usual mathematics)
4. (Indexed) Inductive types: Nat, List, Vec,…
5. Quotient types
6. Quotient Inductive Types: MSet, Int, Mon, …

all while preserving canonicity, consistency and decidability of typing

Implementation is already ongoing, prototype supporting 1-4 by Pujet

The ultimate goal

Once we know OTT+QITs is well-behaved, we can have Rocq with

1. funext: two functions equal iff pointwise equal
2. propext: two propositions equal iff equivalent
3. uip: equality is proof-irrelevant (like in usual mathematics)
4. (Indexed) Inductive types: Nat, List, Vec,…
5. Quotient types
6. Quotient Inductive Types: MSet, Int, Mon, …

all while preserving canonicity, consistency and decidability of typing

Implementation is already ongoing, prototype supporting 1-4 by Pujet

Thank you for your attention!

The universal (finitary) QIT

$$\mathsf{Sig} = \mathsf{record}\ \{\mathsf{C} : \mathsf{Type};\ \mathsf{arity} : \mathsf{C} \to \mathsf{Nat}\}$$

Inductive $\overline{\mathsf{Tm}}\ (\Sigma : \mathsf{Sig})(\Gamma : \mathsf{Type}) : \mathsf{Type} :=$

$\mid \mathsf{var}\ (x : \Gamma) : \overline{\mathsf{Tm}}\ \Sigma\ \Gamma$

$\mid \mathsf{sym}\ (c : \Sigma.\mathsf{C})\ (\mathbf{t} : \mathsf{Vec}\ (\overline{\mathsf{Tm}}\ \Sigma\ \Gamma)\ (\Sigma.\mathsf{arity}\ c)) : \overline{\mathsf{Tm}}\ \Sigma\ \Gamma$

$$\mathsf{EqTh}\ \Sigma = \mathsf{record}\ \{\mathsf{E} : \mathsf{Type};\ \mathsf{Ctx} : \mathsf{E} \to \mathsf{Type};\ \mathsf{lhs}, \mathsf{rhs} : (e : \mathsf{E}) \to \overline{\mathsf{Tm}}\ \Sigma\ (\mathsf{Ctx}\ e)\}$$

Inductive $\mathsf{Tm}\ (\Sigma : \mathsf{Sig})\ (\mathcal{E} : \mathsf{EqTh}\ \Sigma) : \mathsf{Type} :=$

$\mid \mathsf{sym}\ (c : \Sigma.\mathsf{C})\ (\mathbf{t} : \mathsf{Vec}\ (\mathsf{Tm}\ \Sigma\ \mathcal{E})\ (\Sigma.\mathsf{arity}\ c)) : \mathsf{Tm}\ \Sigma\ \mathcal{E}$

$\mid \mathsf{eq}\ (e : \mathcal{E}.\mathsf{E})\ (\gamma : \mathcal{E}.\mathsf{Ctx}\ e \to \mathsf{Tm}\ \Sigma\ \mathcal{E}) : (\mathcal{E}.\mathsf{lhs}\ e)\langle\gamma\rangle = (\mathcal{E}.\mathsf{rhs}\ e)\langle\gamma\rangle$

where $\_\langle\_\rangle : \overline{\mathsf{Tm}}\ \Sigma\ \Gamma \to (\Gamma \to \mathsf{Tm}\ \Sigma\ \mathcal{E}) \to \mathsf{Tm}\ \Sigma\ \mathcal{E}$ is defined by

$(\mathsf{var}\ x)\langle\gamma\rangle := \gamma\ x \qquad (\mathsf{sym}\ c\ [t_1, \ldots, t_k])\langle\gamma\rangle := \mathsf{sym}\ c\ [t_1\langle\gamma\rangle, \ldots, t_k\langle\gamma\rangle]$