# Disornamentation

Lucas Baudin
École Normale Supérieure

Didier Rémy
Gallium, Inria

Paris, France

September 28, 2018

# Ornamentation and disornamentation

```
type expr =
  | Add   of expr * expr
  | Mult  of expr * expr
  | Const of int
```

→

```
type expr' =
  | Binop' of op * expr' * expr'
  | Const' of int

type op = OpAdd | OpMult
```

```
type relation expr_to_expr' : expr => expr' with
  | Add (a, b) => Binop'(OpAdd,  a, b) when a b : expr_to_expr'
  | Mult(a, b) => Binop'(OpMult, a, b) when a b : expr_to_expr'
  | Const i    => Const' i
```

# Ornamentation and disornamentation

```
type expr =
  | Add   of expr * expr
  | Mult  of expr * expr
  | Const of int
```

```
type expr' =
  | Binop' of op * expr' * expr'
  | Const' of int

type op = OpAdd | OpMult
```

```
type relation expr_to_expr' : expr => expr' with
  | Add (a, b) => Binop'(OpAdd,  a, b) when a b : expr_to_expr'
  | Mult(a, b) => Binop'(OpMult, a, b) when a b : expr_to_expr'
  | Const i    => Const' i
```

ornamentation

```
let rec eval e = ...
```

```
let rec eval' e = ...
```

# Ornamentation and disornamentation

```
type expr =
   | Add   of expr * expr
   | Mult  of expr * expr
   | Const of int
```

```
let rec eval e =
  match e with
    | Add(a, b)  →
      plus (eval a) (eval b)
    | Mult(a, b) →
      mult (eval a) (eval b)
    | Const i    →
      i
```

ornamentation →

```
type expr' =
   | Binop' of op * expr' * expr'
   | Const' of int

type op = OpAdd | OpMult
```

```
let rec eval' e =
  match e with
    | Binop'(OpAdd, a, b) →
      plus (eval' a) (eval' b)
    | Binop'(OpMult, a, b) →
      mult (eval' a) (eval' b)
    | Const' i →
      i
```

# Ornamentation and disornamentation

```
type expr =
  | Add   of expr * expr
  | Mult  of expr * expr
  | Const of int
```

```
let rec eval e =
  match e with
    | Add(a, b)  →
       plus (eval a) (eval b)
    | Mult(a, b) →
       mult (eval a) (eval b)
    | Const i    →
       i
```

ornamentation

disornamentation

```
type expr’ =
  | Binop’ of op * expr’ * expr’
  | Const’ of int

type op = OpAdd | OpMult
```

```
let rec eval' e =
  match e with
    | Binop'(OpAdd, a, b) →
       plus (eval' a) (eval' b)
    | Binop'(OpMult, a, b) →
       mult (eval' a) (eval' b)
    | Const' i →
       i
```

# Ornamentation and disornamentation

```
type expr =
  | Add   of expr * expr
  | Mult  of expr * expr
  | Const of int
```

→

```
type expr' =
  | Binop' of op * expr' * expr'
  | Const' of int

type op = OpAdd | OpMult
```

```
type relation expr_to_expr' : expr => expr' with
  | Add(a, b)   => Binop'(OpAdd,  a, b) when a b : expr_to_expr'
  | Mult(a, b)  => Binop'(OpMult, a, b) when a b : expr_to_expr'
  | Const i     => Const' i
```

ornamentation

```
let rec eval e = ...
```

⇄

```
let rec eval' e = ...
```

disornamentation

# Ornamentation and disornamentation

```
type expr =
  | Add   of expr * expr
  | Mult  of expr * expr
  | Co
```

```
type expr' =
  | Binop' of op * expr' * expr'
  | Const' of int
```

- Pierre-Evariste Dagand and Conor McBride.
  « Transporting Functions across Ornaments » ICFP 2012
- Hsiang-Shang Ko and Jeremy Gibbons.
  « Programming with Ornaments » JFP 2012
- Thomas Williams and Didier Rémy.
  « A Principled approach to Ornamentation in ML » POPL 2018

ornamentation

```
let rec eval e = ...
```

```
let rec eval' e = ...
```

disornamentation

# Ornamentation and disornamentation

```
type expr =
  | Add   of expr * expr
  | Mult  of expr * expr
  | Const of int
```

→

```
type expr' =
  | Binop' of op * expr' * expr'
  | Const' of int

type op = OpAdd | OpMult
```

```
type relation expr_to_expr' : expr => expr' with
  | Add(a, b)  => Binop'(OpAdd,  a, b) when a b : expr_to_expr'
  | Mult(a, b) => Binop'(OpMult, a, b) when a b : expr_to_expr'
  | Const i    => Const' i
```

ornamentation

```
let rec eval e = ...
```

⇄

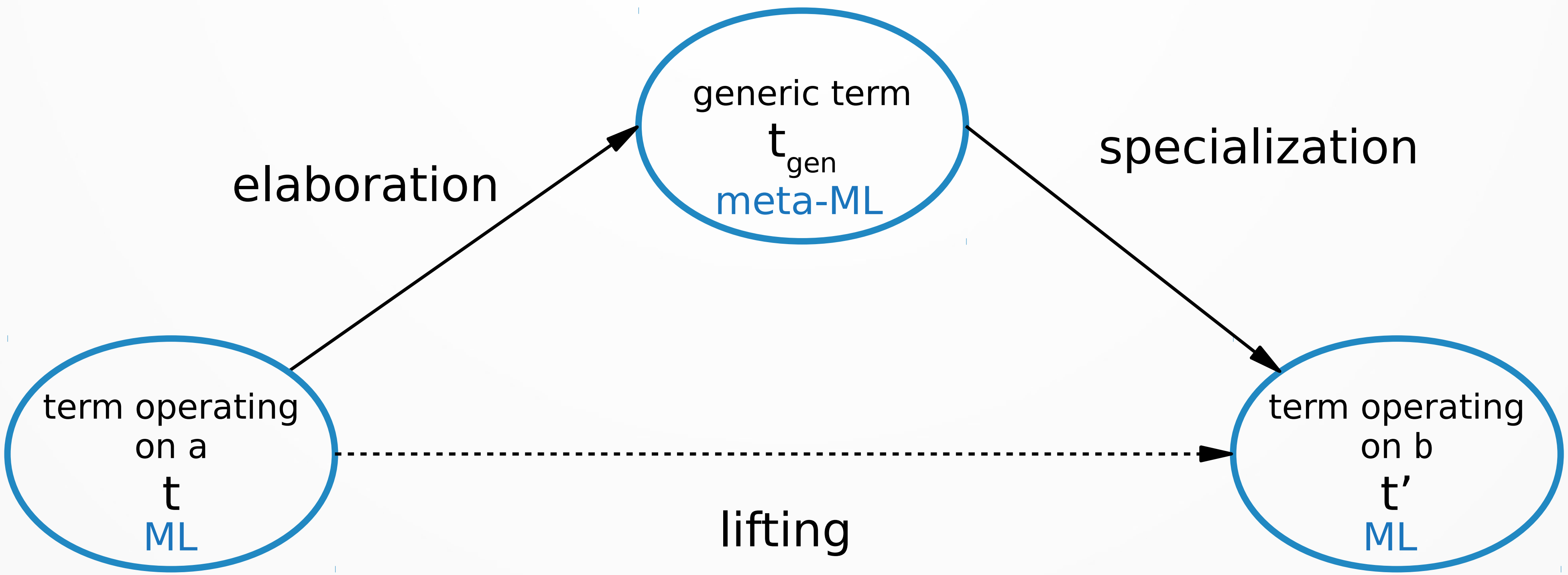
```
let rec eval' e = ...
```

disornamentation

# Demo

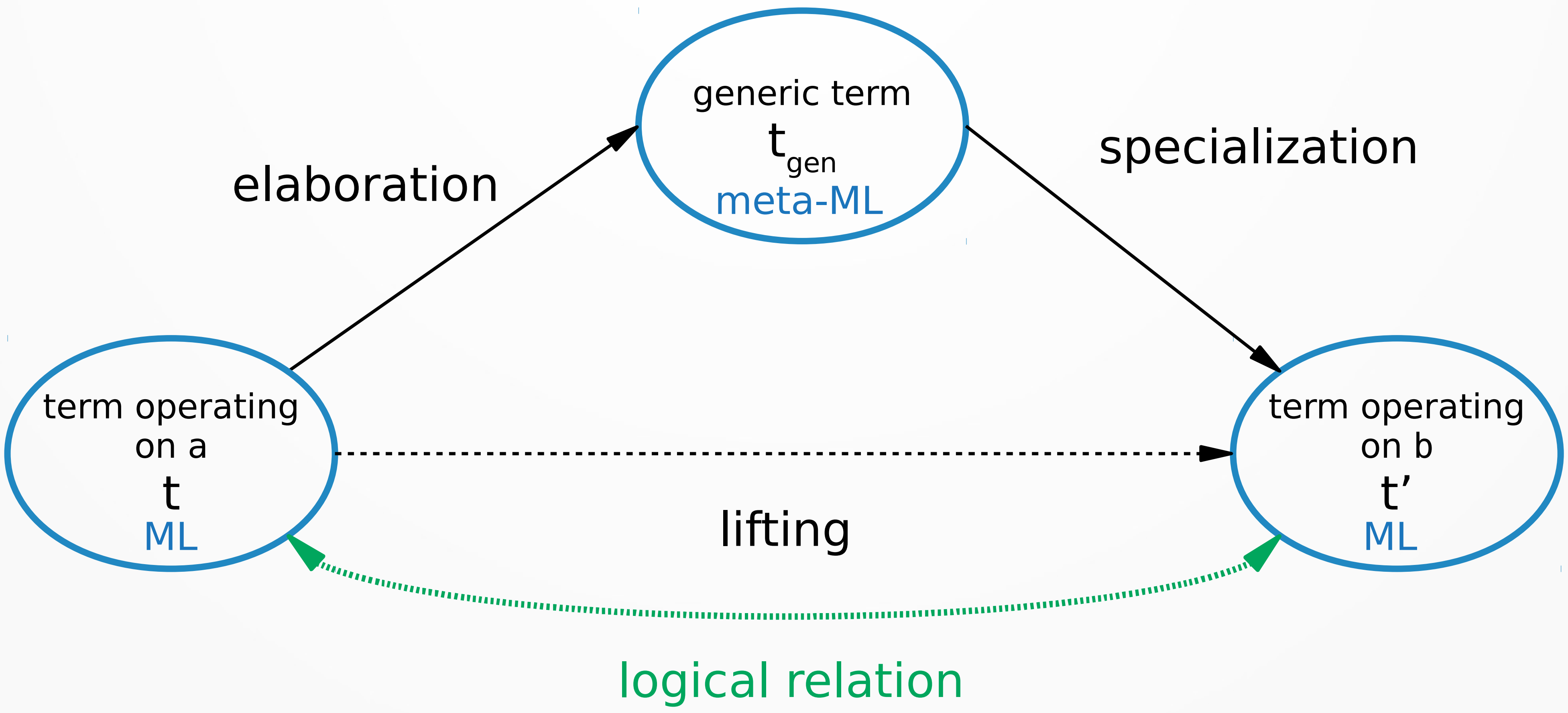
- Examples:
  http://gallium.inria.fr/~remy/ornaments/disorn/
- Try the online prototype:
  https://www.eleves.ens.fr/home/lbaudin/demo

# Transformation
## Two steps



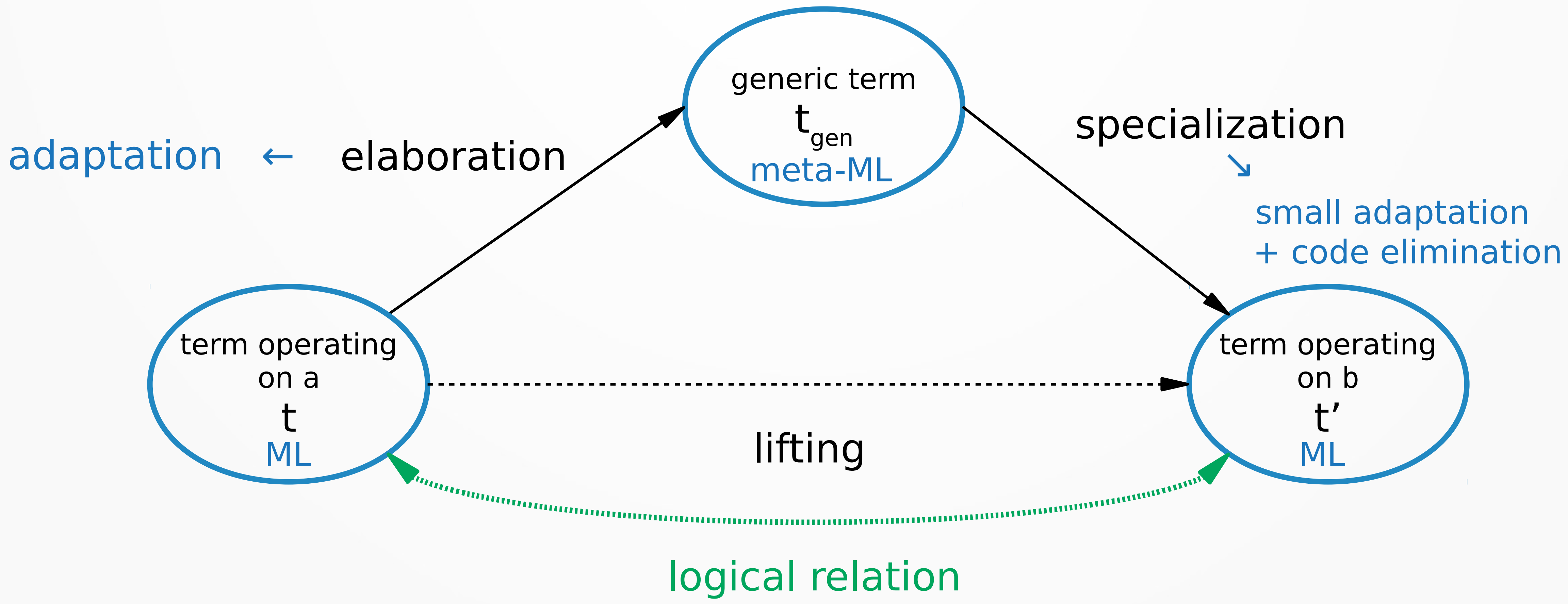
# Transformation

## Two steps



# Transformation

## Two steps



# Generic term

- skeleton type:

```
type nat = Z | S of nat | Zero of empty
type (α, β) nat_skel = Z_skel | S_skel of α | Zero_skel of β
```

# Generic term

- skeleton type:

```
type nat = Z | S of nat | Zero of empty
type (α, β) nat_skel = Z_skel | S_skel of α | Zero_skel of β
```

```
let rec length n = match n with
    | Nil -> Z
    | Cons(_, b) -> S (length b)
```

elaboration →

```
fun rel_1 rel_2 →
let rec length_gen n =
  match rel_1.to_skel n #7 with
   | Nil_skel →
     rel_2.from_skel Z_skel #5
   | Cons_skel(_, b) →
     rel_2.from_skel
        (S_skel (length_gen b))
        #3
```

# Generic term

- skeleton type:

```
type nat = Z | S of nat | Zero of empty
type (α, β) nat_skel = Z_skel | S_skel of α | Zero_skel of β
```

```
let rec length n = match n with
    | Nil -> Z
    | Cons(_, b) -> S (length b)
```

elaboration

```
fun rel₁ rel₂ →
let rec length_gen n =
  match rel₁.to_skel n #7 with
   | Nil_skel →
     rel₂.from_skel Z_skel #5
   | Cons_skel(_, b) →
     rel₂.from_skel
        (S_skel (length_gen b))
        #3
```

- relation encoding : $\texttt{rel} : (\delta^{\text{from\_skel}}, \delta^{\text{to\_skel}}, \text{from\_skel}, \text{to\_skel})$

# Generic term

- skeleton type:

```
type nat = Z | S of nat | Zero of empty
type (α, β) nat_skel = Z_skel | S_skel of α | Zero_skel of β
```

```
let rec length n = match n with
    | Nil -> Z
    | Cons(_, b) -> S (length b)
```

elaboration

```
fun rel₁ rel₂ →
let rec length_gen n =
  match rel₁.to_skel n #7 with
   | Nil_skel →
     rel₂.from_skel Z_skel #5
   | Cons_skel(_, b) →
     rel₂.from_skel
        (S_skel (length_gen b))
        #3
```

- relation encoding : $\texttt{rel} : (\delta^{\text{from\_skel}}, \delta^{\text{to\_skel}}, \text{from\_skel}, \text{to\_skel})$

extension type

conversion functions

# Encoding relations

- example for α `natlist` :

```
type relation α natlist : nat => α list with
    | Z   => Nil
    | S n => Cons(_, n) when n : α natlist

to_skel = λx. λ(). match x with                δ^to_skel = λ _. unit
    | Nil         → Z_skel
    | Cons(_, n) → S_skel n

from_skel = λx. λy. match x with               δ^from_skel = λx. match x with
    | Z_skel       → Nil                            | Z_skel       → unit
    | S_skel n     → Cons(y, n)                     | S_skel n     → α
    | Zero_skel x → x                               | Zero_skel x → σ
```

# Encoding relations

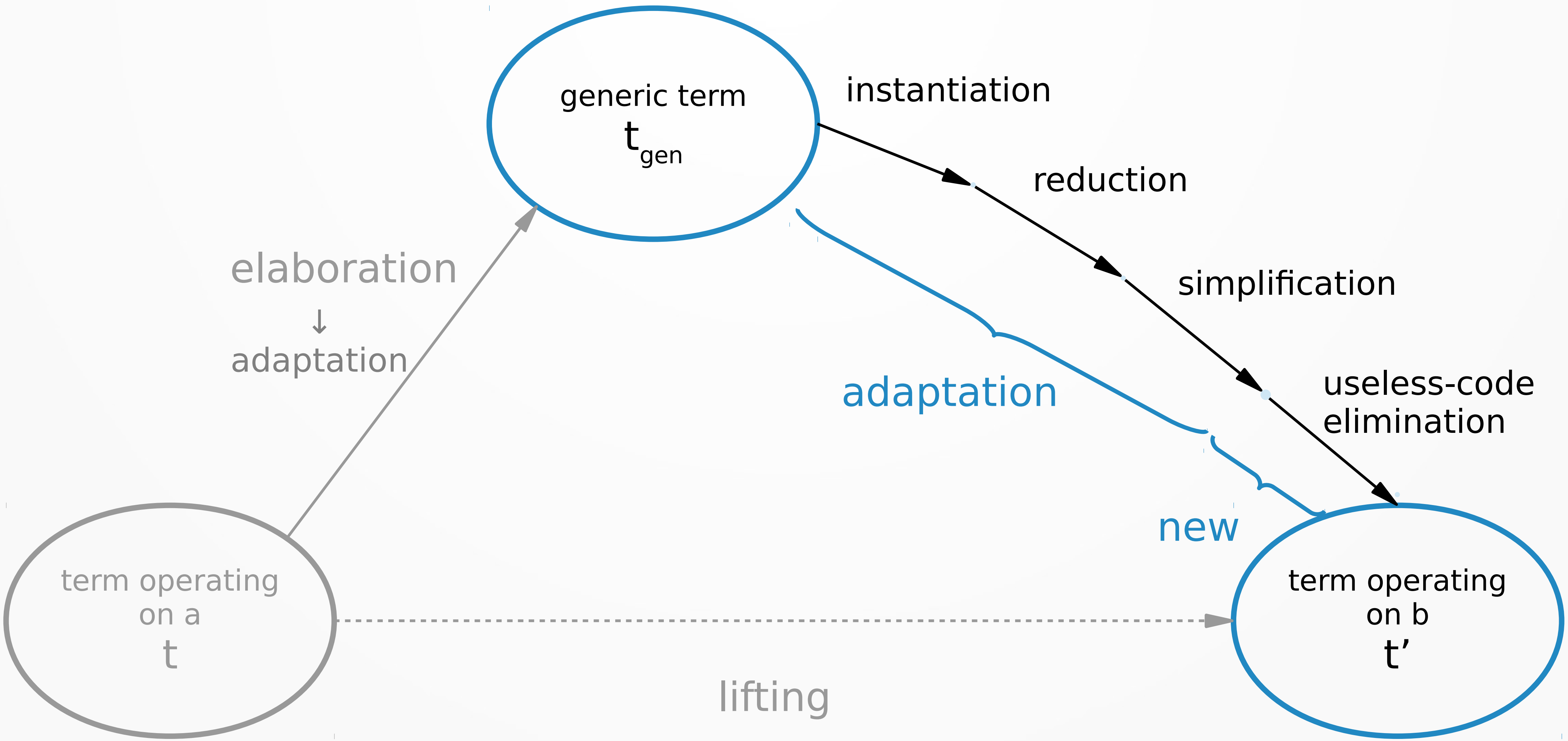
- example for α `rev_natlist :`

```
type relation α rev_natlist : α list => nat with
    | Nil         => Z
    | Cons(_, n)  => S n when n : α natlist

to_skel = λx. λy. match x with
    | Z   → Nil_skel
    | S n → Cons_skel(y, n)

from_skel = λx. λ(). match x with
    | Nil_skel          → Z
    | Cons_skel(_, n)   → S n
    | Zero_skel x       → x
```

```
δto_skel = λx. match x with
    | Z   → unit
    | S n → α

δfrom_skel = λ x. match x with
    | Nil_skel      → unit
    | Cons_skel     → unit
    | Zero_skel x → σ
```

Transformation

# Specialization



# Useless-code elimination

- ornamentation: add pieces of information, no code need to be removed
- disornamentation: code used to compute removed pieces of information becomes useless

```
let rec map f l =
   match l with
   | Nil → Nil
   | Cons(a, q) →
      let a’ = f a in
      Cons(a’, map f q)
```

disornamentation →

```
let rec id f l =
   match l with
   | Z → Z
   | S q →
      let a’ = f #3 in
      S (id f q)
```

useless

# Motivation

- for ornamentation, holes were numbered

```
let concat' = lifting add : ...

let rec concat' m n = match m with
    | Nil → n
    | Cons(_, m') → Cons(#3, concat' m' n)
```

```
let concat' = lifting add : ... with
 patch #3[match m with Cons(a,_) → a]

let rec concat' m n = match m with
    | Nil → n
    | Cons(a, m') → Cons(a, concat' m' n)
```

  - not robust to changes
  - does not allow easy patch factorization
  - cannot be extended to new, similar holes
- problem already noticed with ornamentation, but not solved

# Patches

- a patch is composed of:
  - a pattern, *i.e.* a term with metavariables and a unique hole denoted #[...]
  - a term, the patch content

```
let concat = lifting add : ... with
   patch match _ with  Cons(a, m') -> Cons(#[a], _)
```

# Patches

```
let rec concat' m n = match m with
      | Nil → n
      | Cons(_, m') → Cons(#3, concat' m' n)
```

```
let concat = lifting add : ... with
    patch match _ with  Cons(a, m') -> Cons(#[a], _)
```

pattern written by the user

content

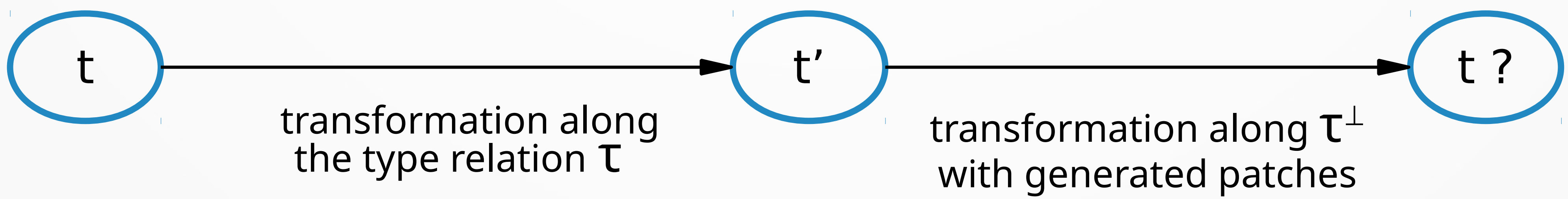
$$p = C\gamma D[\sharp[a]]$$

toplevel pattern

```
fun m n → []
```

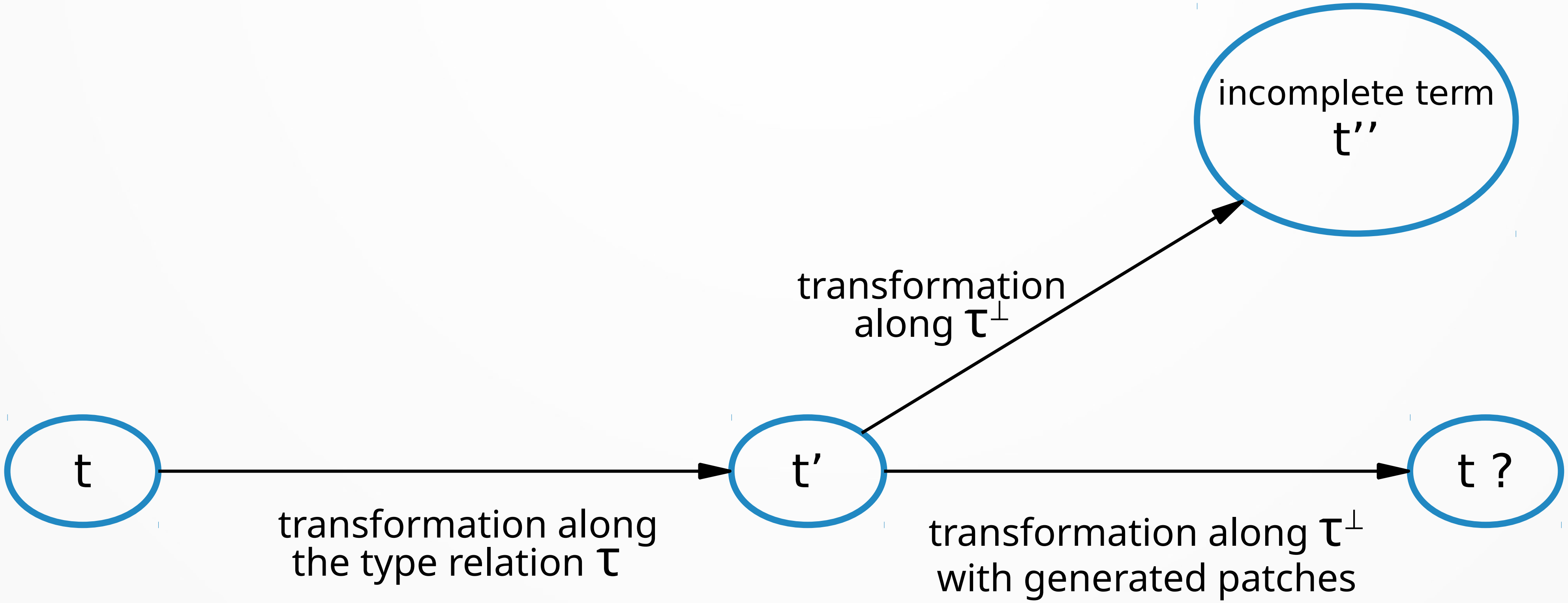
pattern variable `[]`

# Generation


t
t’
t ?
transformation along
the type relation τ
transformation along τ⊥
with generated patches

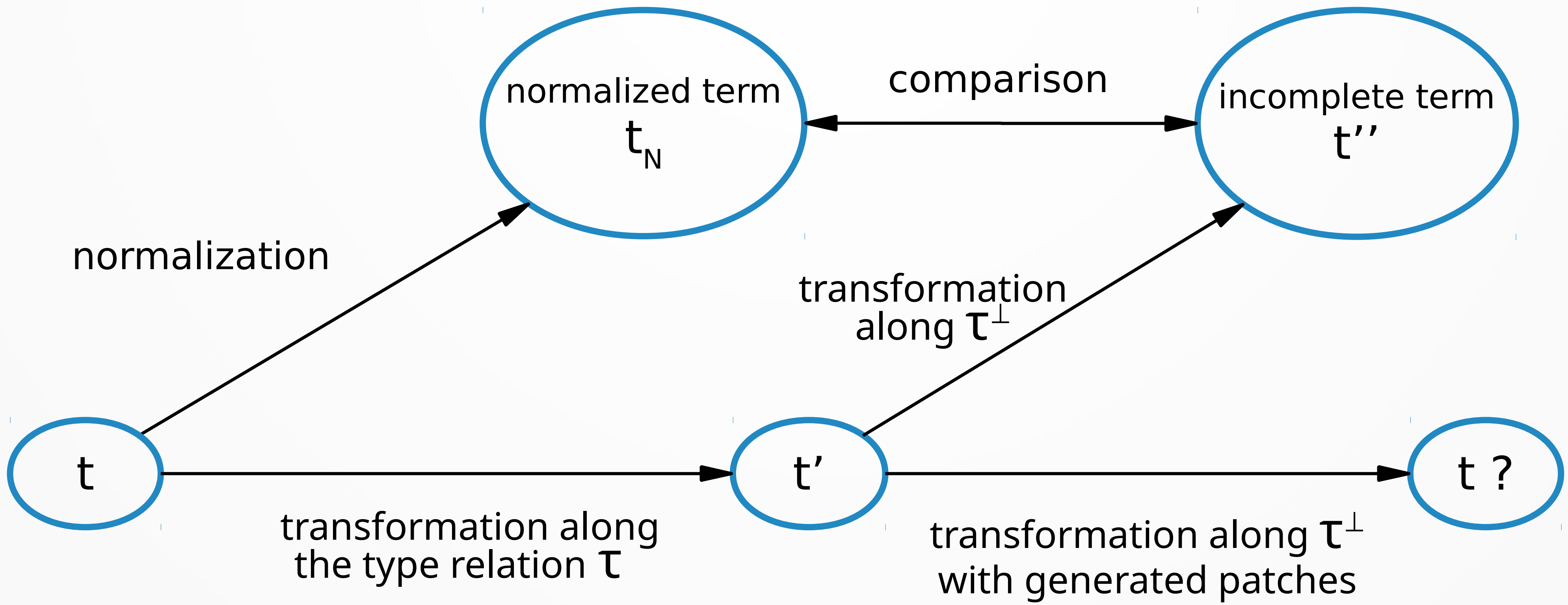
## A new patch language

# Generation



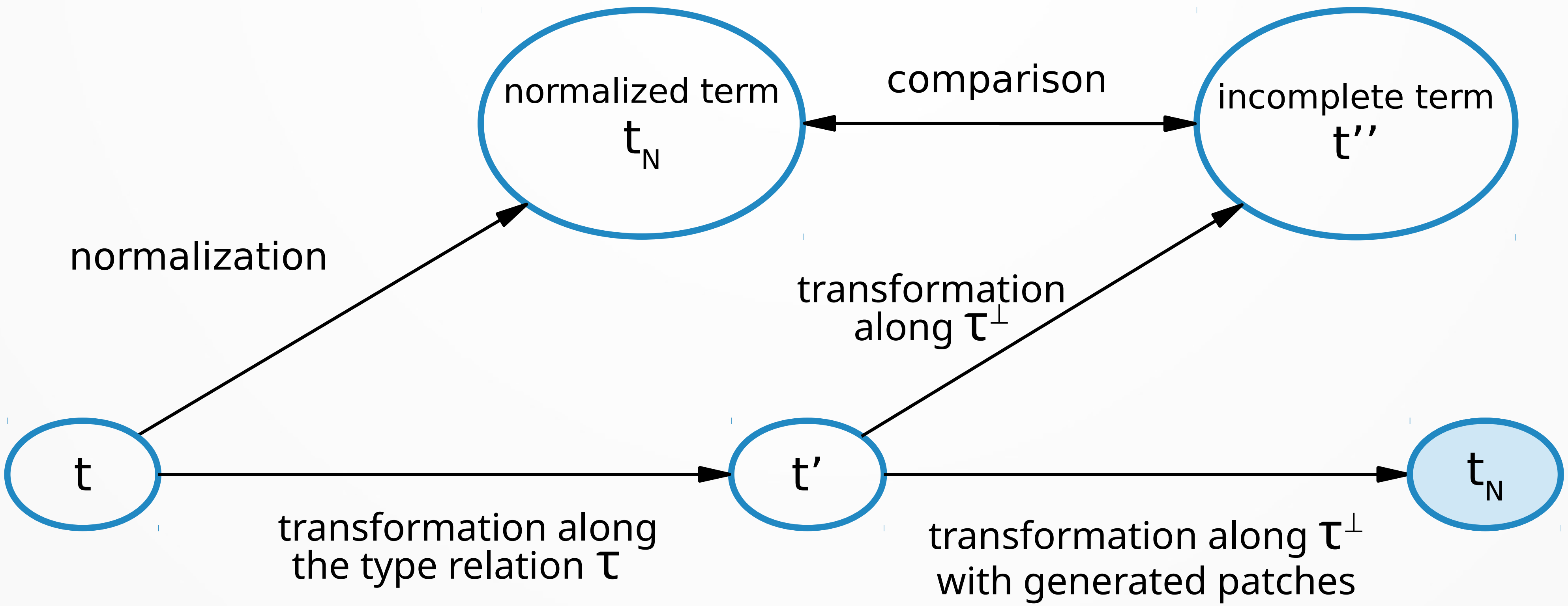
A new patch language

# Generation



A new patch language

# Generation



# Perspectives

- prove more properties, for instance:
  - syntactic conservation on terms that are ornamented and then disornamented
  - completeness of patch generation
- study more transformations: add/remove function arguments, research on the generic term
- OCaml implementation: ongoing work for ornamentatoin

# Bring back home

- both theory and implementation of ornamentation reused
- a new, more general framework to express both disornamentation and ornamentation
- ornamentation and disornamentation → code synchronization
  - a new patch language
  - patch generation

# Relation logique

- relation logique (step-indexed) $\mathcal{V}[\tau]_\gamma$
- généralise les types avec des relations

$$\tau ::= \ldots \mid \chi\,(\tau)^i$$

- étendue pour les relations, par exemple :

$$\mathcal{V}[\alpha\ \textsf{natlist}]_\gamma = \{(\textsf{Z}, \textsf{Nil})\} \cup \{(\textsf{S}\ \textsf{Z}, \textsf{Cons}(x, \textsf{Nil})) \mid x : \alpha\} \cup \ldots$$

- correction de la transformation
  - **Théorème** : si a est transformé en A le long d'un type relation τ alors

$$(a, A) \in \mathcal{V}[\tau]_\gamma$$

# Travaux connexes

- transformation sur les données :
  - Foster, J. Nathan, Alexandre Pilkiewicz, et Benjamin C. Pierce. « Quotient Lenses ». In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, 383–396. ICFP ’08. New York, NY, USA: ACM, 2008. https://doi.org/10.1145/1411204.1411257.
- ornementation pour ML :
  - Williams, Thomas, et Didier Rémy. « A Principled Approach to Ornamentation in ML - Long ». *Proceedings of the ACM on Programming Languages* 2, no POPL (27 décembre 2017): 1-30. https://doi.org/10.1145/3158109.
- ornementation en Agda :
  - Dagand, Pierre-Evariste, et Conor McBride. « Transporting Functions across Ornaments », 103. ACM Press, 2012. https://doi.org/10.1145/2364527.2364544.
  - KO, HSIANG-SHANG, et JEREMY GIBBONS. « Programming with Ornaments », 2016, 42.
- langage de patchs :
  - Andersen, Jesper, et Julia Lawall. « Generic Patch Inference », 20 juin 2018. http://coccinelle.lip6.fr/papers/andersen_ase08.pdf.