Disornamentation

Ornamentation as it should have been

Lucas Baudin
École Normale Supérieure

Didier Rémy
Inria

Paris, France

September 28, 2018

Changes in data structures

- data structures of existing programs sometimes need to be modified
- the existing pieces of code must be modified accordingly
  - doing so by hand: tedious, incomplete, and error prone
- ornamentation: a class of transformations in data-structures for which the code can be semi-automatically adapted.
  - a form of refactoring

Ornamentation

```
type exp =
  | App of exp * exp
  | Abs of (exp -> exp)
  | Const of int
```

```
type exploc =
  | App' of exploc * exploc * loc
  | Abs' of (exploc -> exploc) * loc
  | Const' of int * loc
```

Ornamentation

```
type exp =
  | App of exp * exp
  | Abs of (exp -> exp)
  | Const of int
```

```
type exploc =
  | App' of exploc * exploc * loc
  | Abs' of (exploc -> exploc) * loc
  | Const' of int * loc
```

```
type relation add_loc : exp => exploc with
  | App (u, v) => App' (u, v, _) when u v : add_loc
  | Abs f      => Abs' (f, _)    when f : add_loc -> add_loc
  | Const i    => Const' (i, _)
```

Ornamentation

```
type exp =
  | App of exp * exp
  | Abs of (exp -> exp)
  | Const of int
```

```
type exploc =
  | App' of exploc * exploc * loc
  | Abs' of (exploc -> exploc) * loc
  | Const' of int * loc
```

```
type relation add_loc : exp => exploc with
  | App (u, v) => App' (u, v, _) when u v : add_loc
  | Abs f      => Abs' (f, _)    when f : add_loc -> add_loc
  | Const i    => Const' (i, _)
```

Ornamentation

```
type exp =
  | App of exp * exp
  | Abs of (exp -> exp)
  | Const of int
```

```
type exploc =
  | App' of exploc * exploc * loc
  | Abs' of (exploc -> exploc) * loc
  | Const' of int * loc
```

```
type relation add_loc : exp => exploc with
  | App (u, v) => App' (u, v, _) when u v : add_loc
  | Abs f      => Abs' (f, _)    when f : add_loc -> add_loc
  | Const i    => Const' (i, _)
```

ornamentation

```
eval : exp → ...
```

```
eval’ : exploc → ...
```

Ornamentation and disornamentation

- ornamentation:
  - add pieces of data
  - reorganize constructors
  - rename constructors
  - remove constructors

Ornamentation and disornamentation

- ornamentation:
  - add pieces of data
  - reorganize constructors
  - rename constructors
  - remove constructors
- disornamentation:
  - remove pieces of data
  - reorganize constructors
  - rename constructors
  - add constructors

Ornamentation and disornamentation

- ornamentation: 
  - add pieces of data
  - reorganize constructors
  - rename constructors
  - remove constructors
- disornamentation: ?
  - remove pieces of data
  - reorganize constructors
  - rename constructors
  - add constructors

Ornamentation and disornamentation

- ornamentation: ✓
  - add pieces of data
  - reorganize constructors
  - rename constructors
  - remove constructors
- disornamentation: ?
  - remove pieces of data
  - reorganize constructors
  - rename constructors
  - add constructors

- relations = ornaments + disornaments
  - extension of the ornamentation framework!

Ornamentation and disornamentation

- ornamentation: ✓
  - add pieces of data
  - reorganize constructors
  - re
  - re
- disornamentation: ?
  - remove pieces of data
  - reorganize constructors
- relat
  - ex

- idea of Conor McBride with Pierre-Evariste Dagand
- also studied by Hsiang-Shang Ko and Jeremy Gibbons
- based on work by Thomas Williams and Didier Rémy that adapted the idea of ornaments to ML

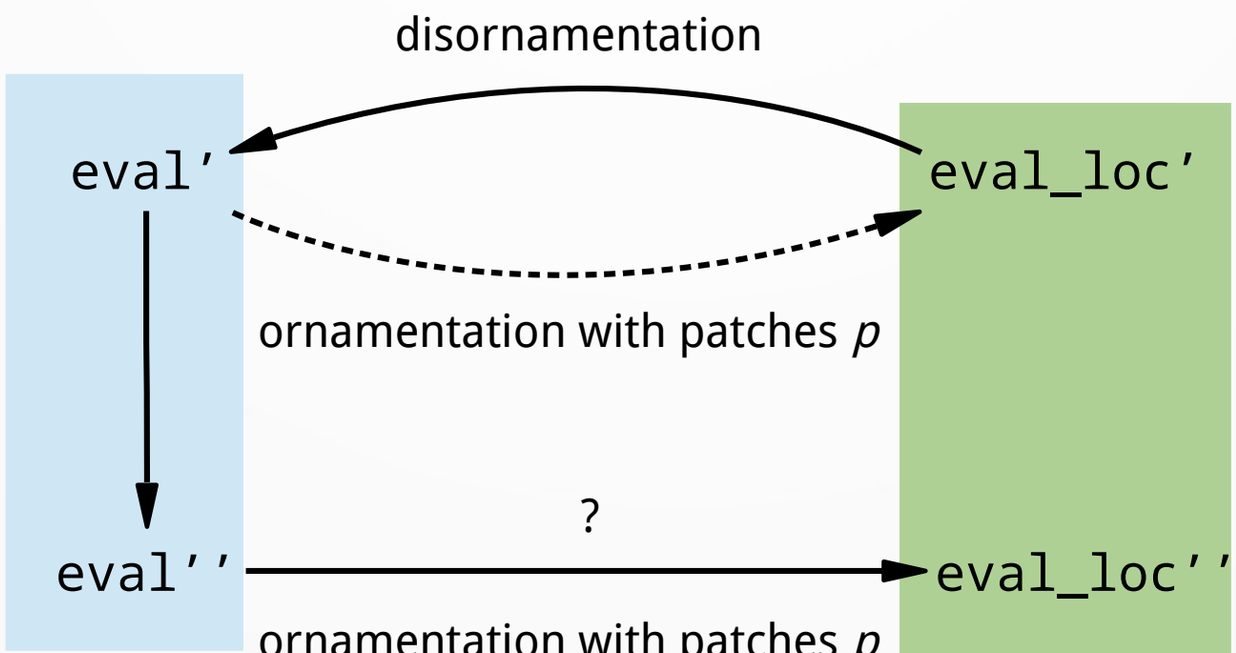
Synchronization: two views of the same algorithm

code simpler,
no error handling

code bigger,
error handling but less readable

```
type exp
```

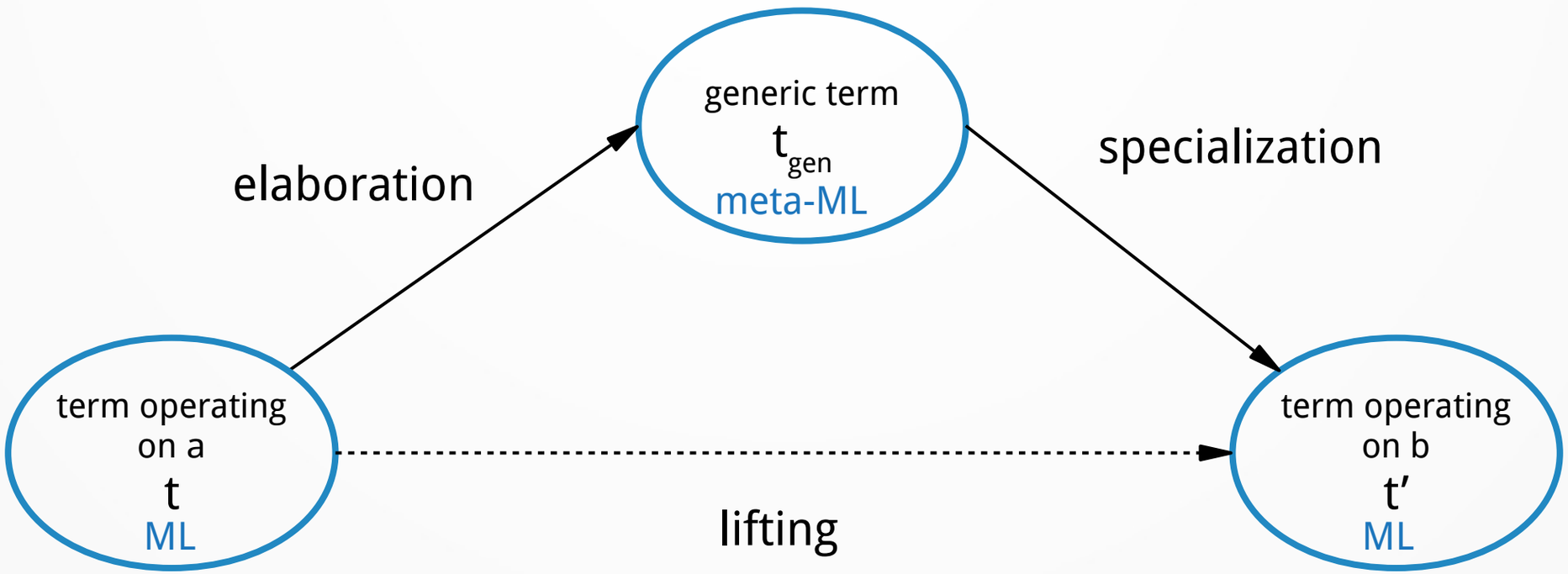
```
type exploc
```



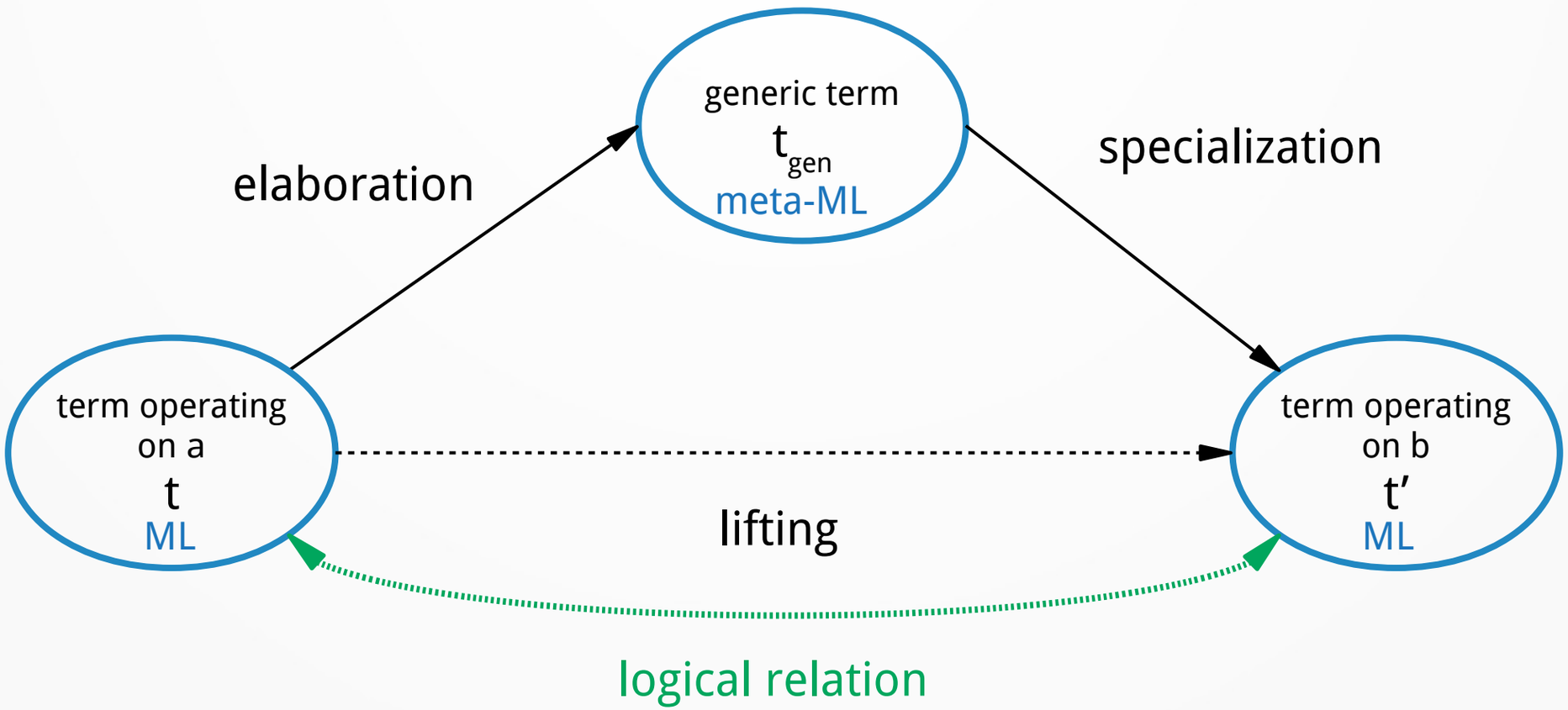
Demo

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

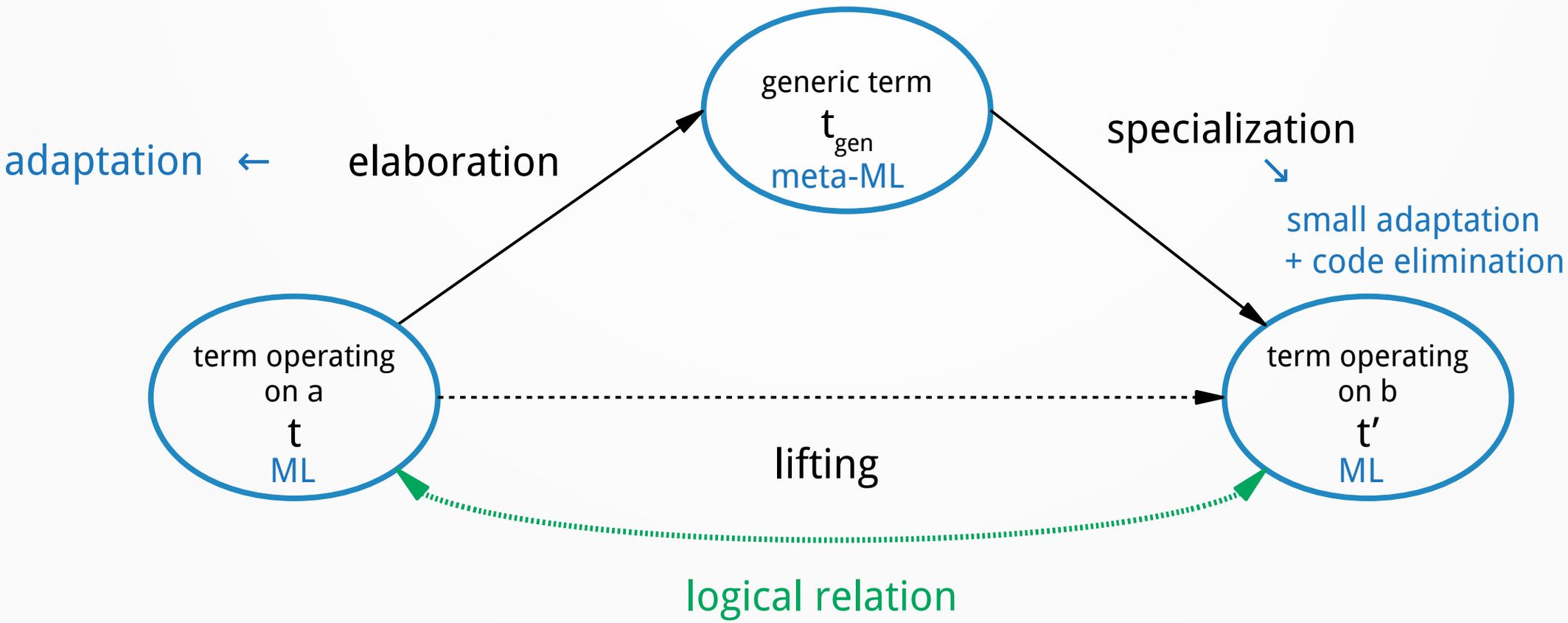
Framework


generic term
t_{gen}
meta-ML
elaboration
specialization
term operating
on a
t
ML
lifting
term operating
on b
t'
ML

Framework


generic term
t_{gen}
meta-ML
elaboration
specialization
term operating
on a
t
ML
term operating
on b
t'
ML
lifting
logical relation

Framework


generic term
t_{gen}
meta-ML
adaptation ←
elaboration
specialization
↘
small adaptation
+ code elimination
term operating
on a
t
ML
term operating
on b
t'
ML
lifting
logical relation

Transformation

Generic term

- skeleton type:

```
type α list           = Nil       | Cons of α * α list
type (α, β) list_skel = Nil_skel  | Cons_skel of α * β
```

Generic term

- skeleton type:

```
type α list           = Nil       | Cons of α * α list
type (α, β) list_skel = Nil_skel  | Cons_skel of α * β
```

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


```
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
```

Generic term

- skeleton type:

```
type α list           = Nil      | Cons of α * α list
type (α, β) list_skel = Nil_skel | Cons_skel of α * 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 : `rel` : $(\delta^{\text{from\_skel}}, \delta^{\text{to\_skel}}, \text{from\_skel}, \text{to\_skel})$

Generic term

- skeleton type:

```
type α list           = Nil      | Cons of α * α list
type (α, β) list_skel = Nil_skel | Cons_skel of α * 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 : $\texttt{rel} : (\delta^{\text{from\_skel}}, \delta^{\text{to\_skel}}, \text{from\_skel}, \text{to\_skel})$

type functions that describe the type of extra information

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
    | Nil        → Z_skel
    | Cons(_, n) → S_skel n

from_skel = λx. λy. match x with
    | Z_skel      → Nil
    | S_skel n    → Cons(y, n)
```

$\delta^{\texttt{to\_skel}}$ = λ _. unit

```
δ^from_skel = λx. match x with
    | Z_skel      → unit
    | S_skel n    → α
```

Encoding relations

- example for α `rev_natlist` :

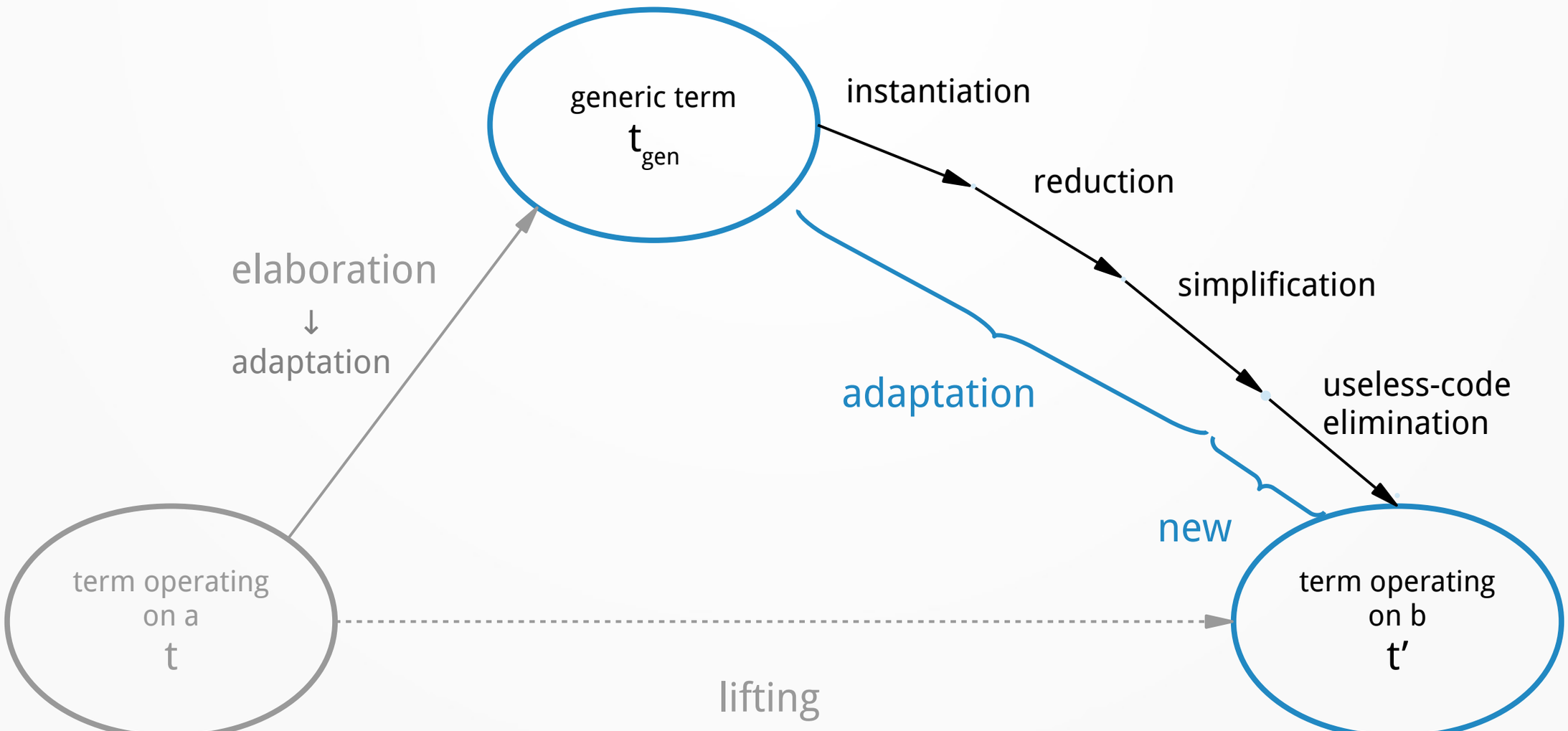
```
type relation α rev_natlist : α list => nat with
    | Nil         => Z
    | Cons(_, n) => S n when n : α natlist

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
```

$\delta^{\texttt{to\_skel}}$ = λx. match x with
    | Z → unit
    | S n → α

$\delta^{\texttt{from\_skel}}$ = λ x. match x with
    | Nil_skel → unit
    | Cons_skel → unit

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

A new patch language

- 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

A new patch language

- 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], _)
```

A new patch language

```
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

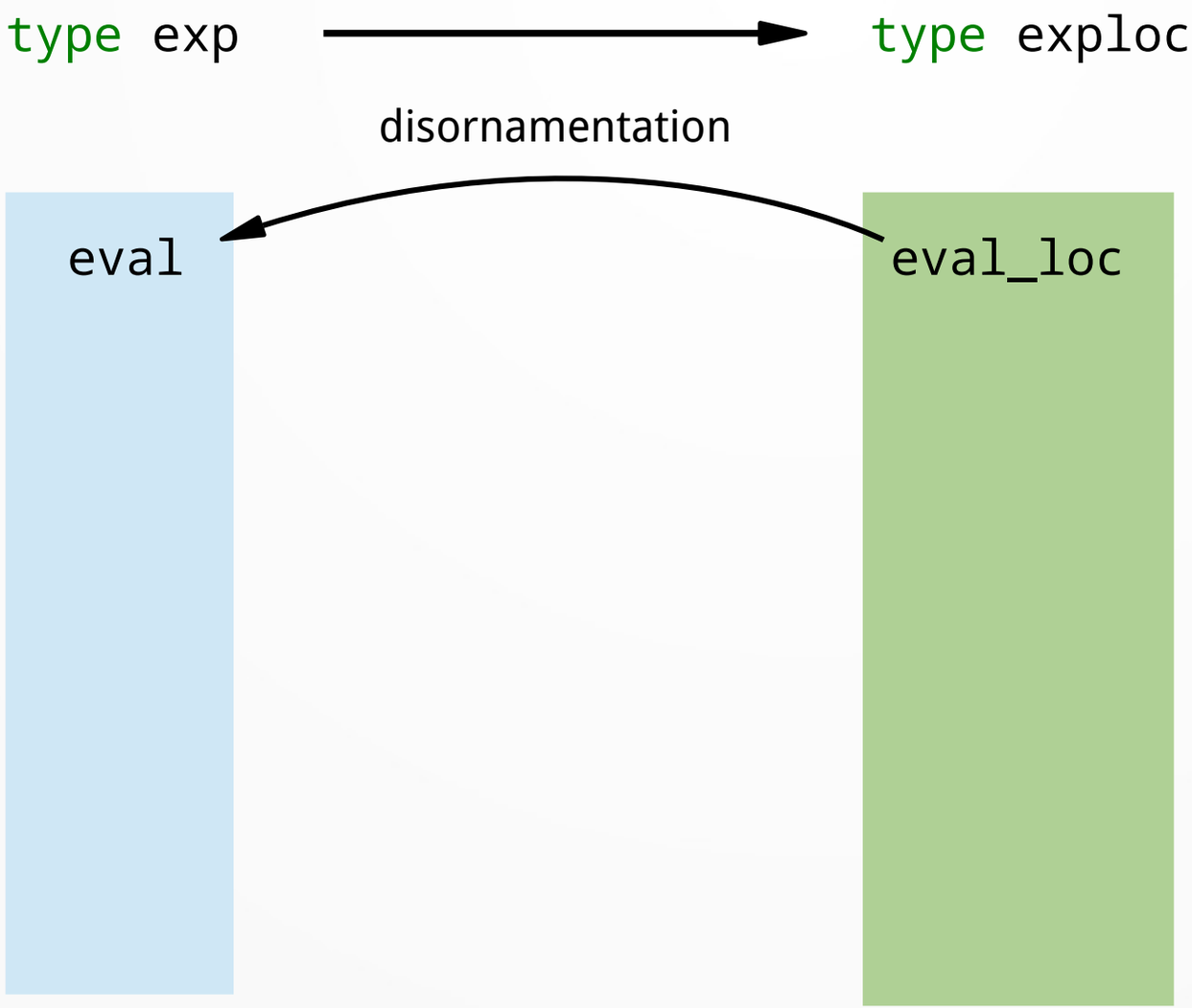
$$p = C\gamma D[\sharp[a]]$$

toplevel pattern

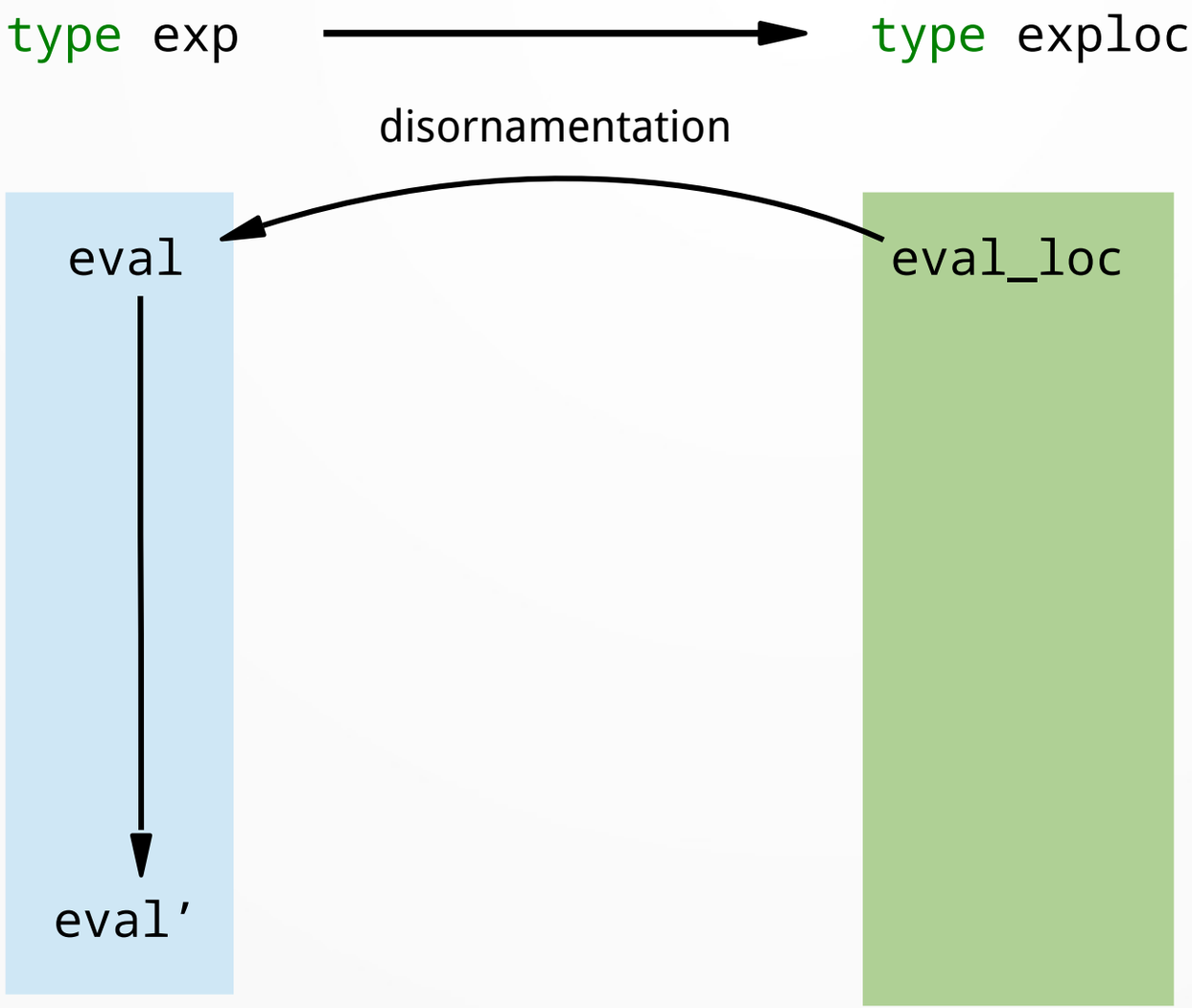
```
fun m n → []
```

pattern variable `[]`

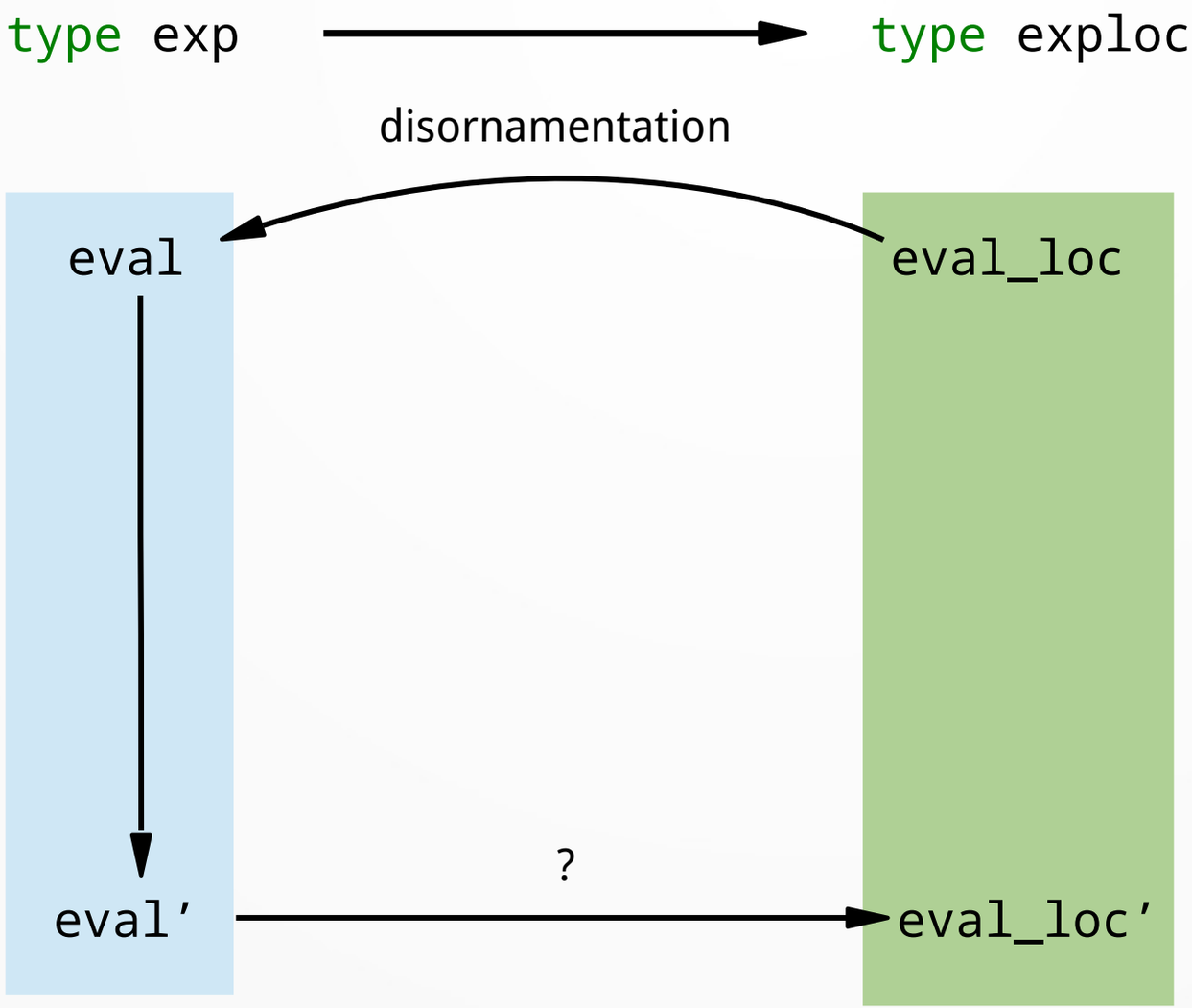
Generation



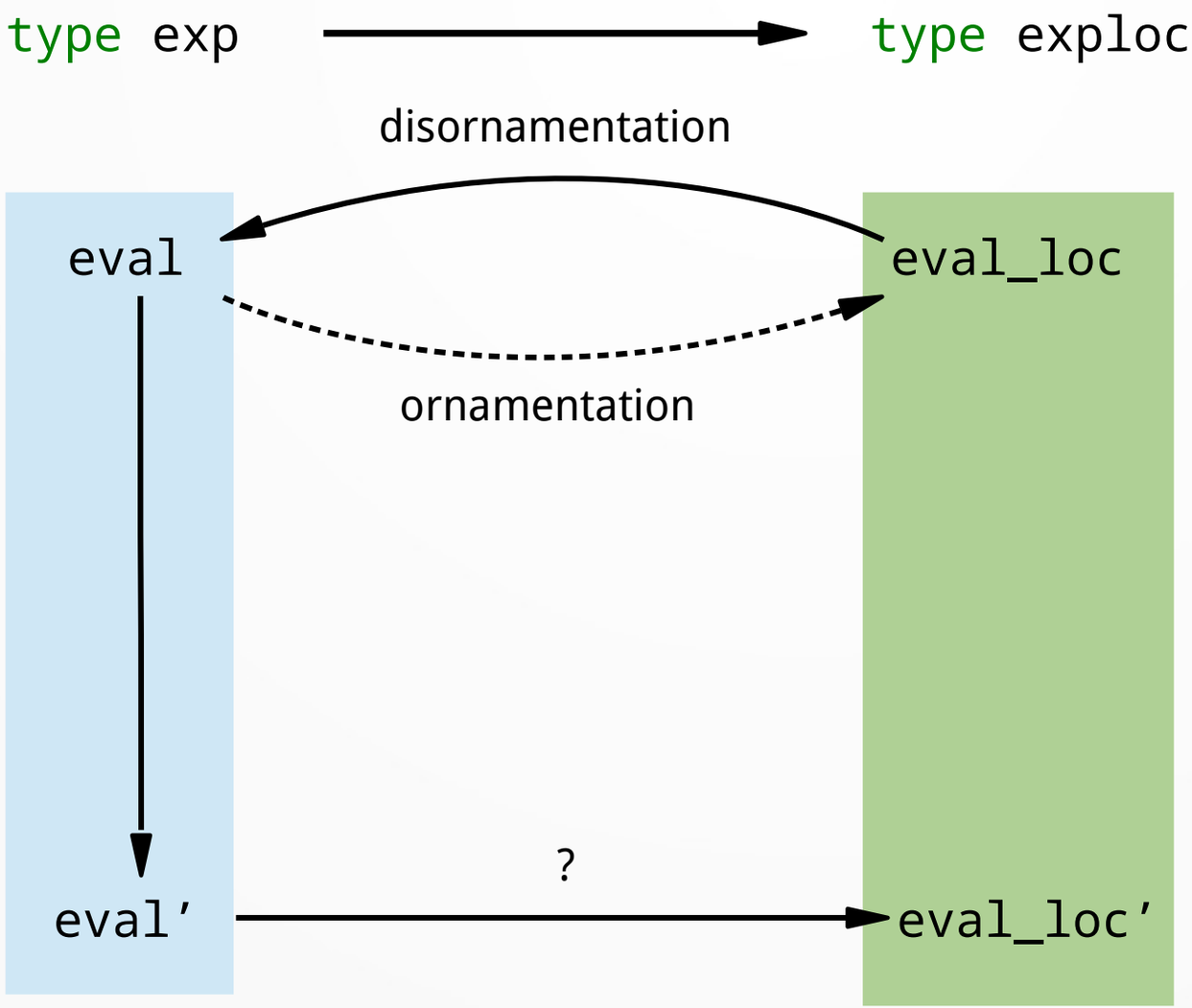
Generation



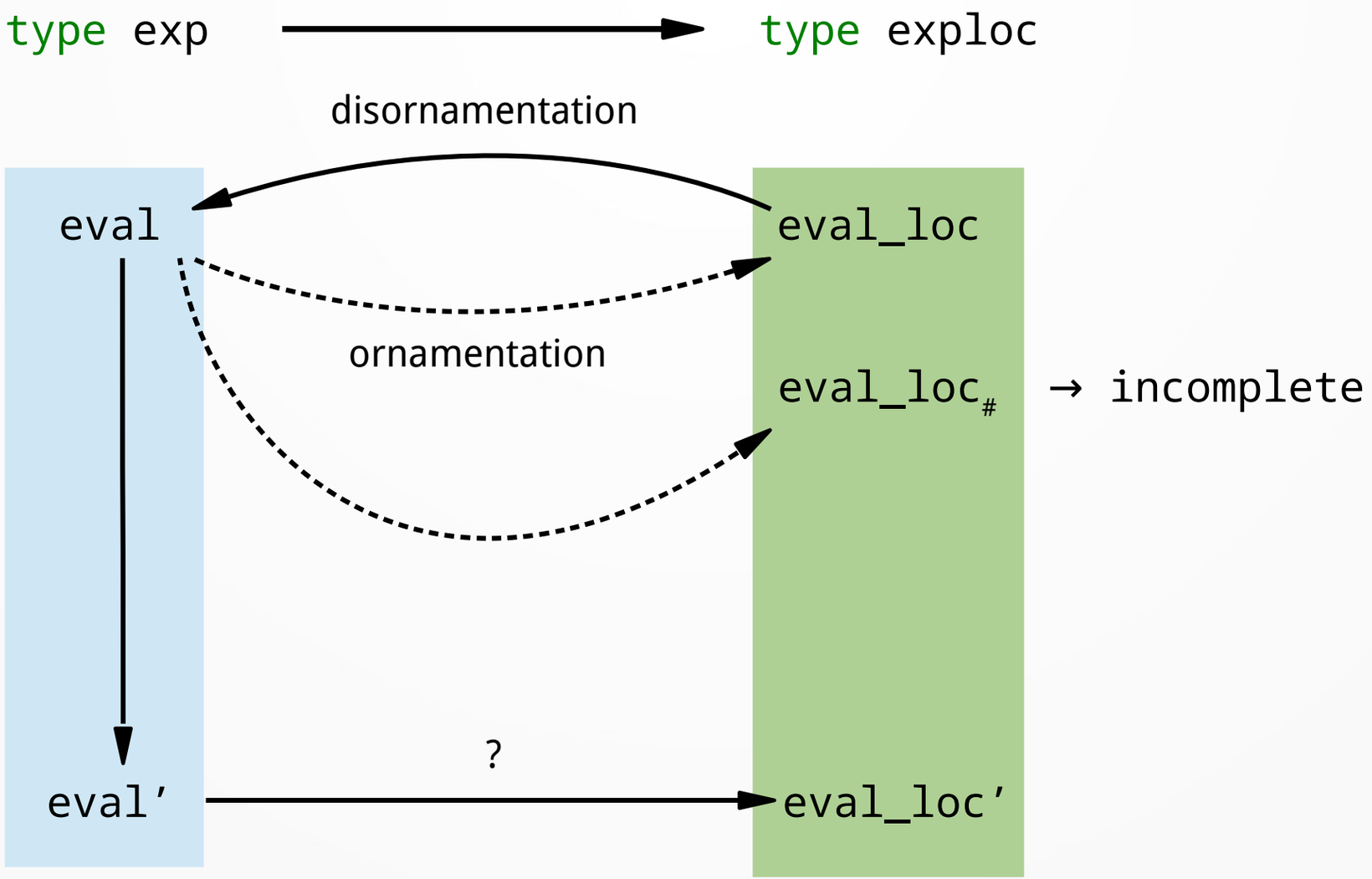
Generation


type exp
type exploc
disornamentation
eval
eval_loc
eval'
?
eval_loc'

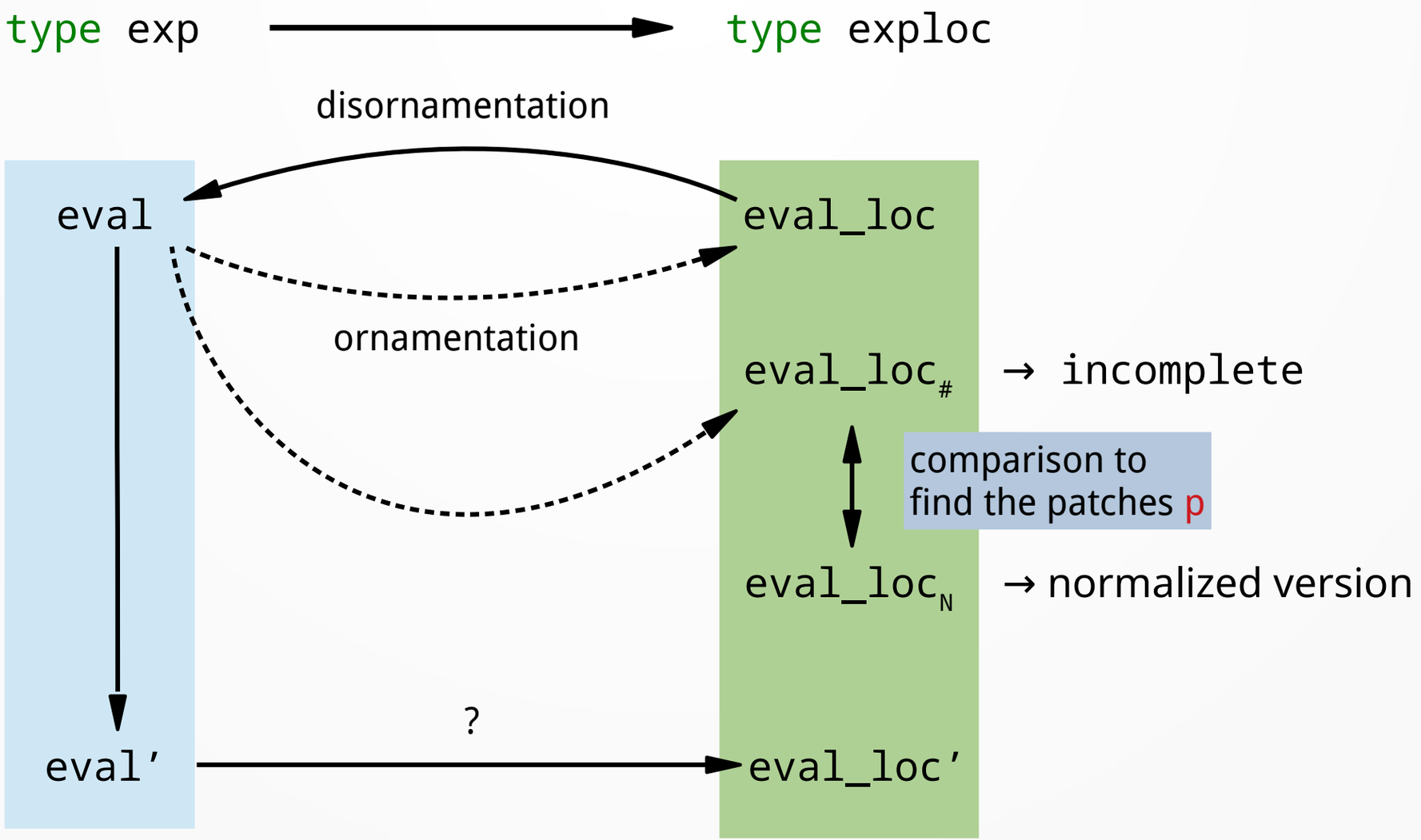
Generation


type exp
type exploc
disornamentation
eval
eval_loc
ornamentation
?
eval'
eval_loc'

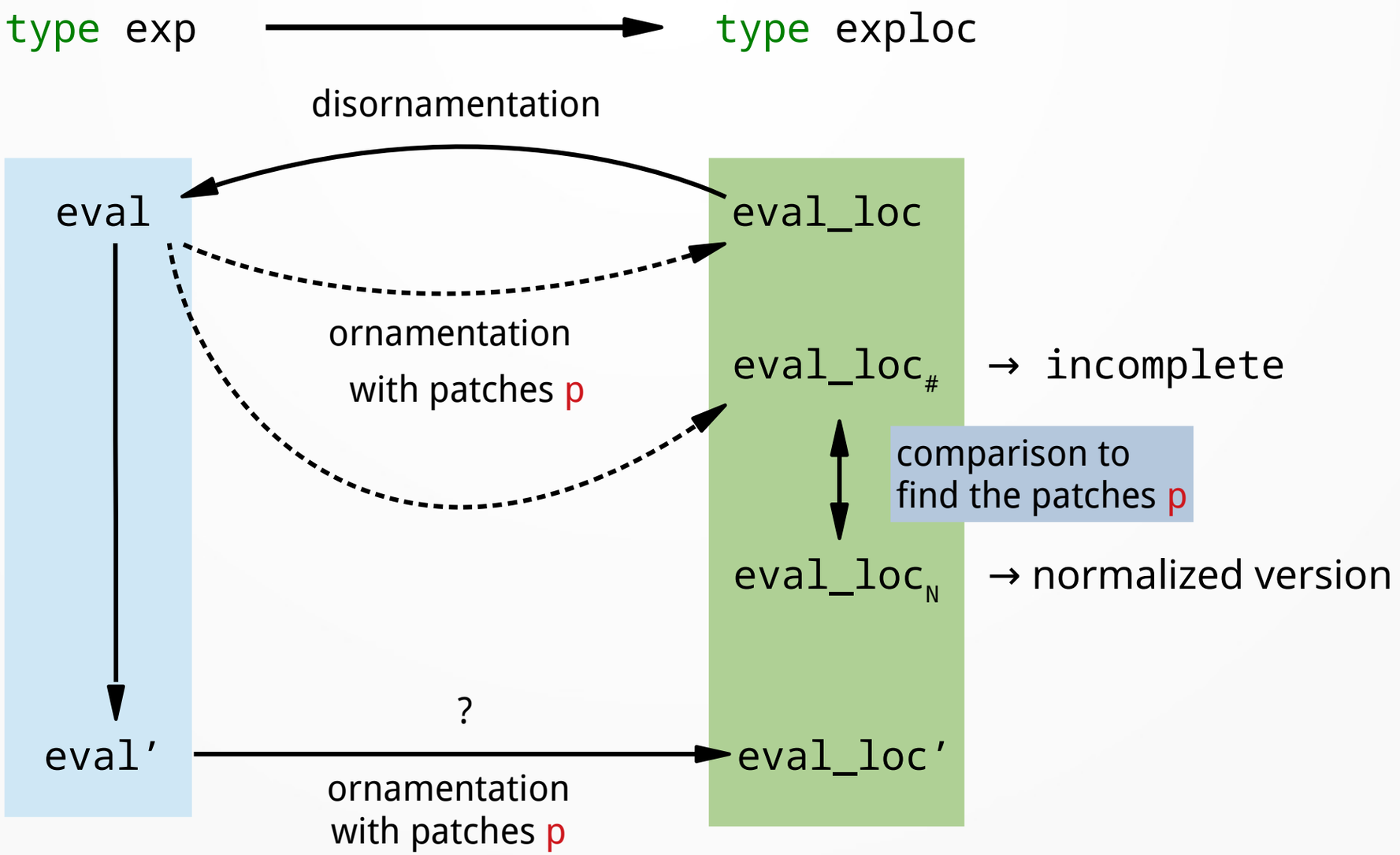
Generation


type exp
type exploc
disornamentation
eval
eval_loc
ornamentation
eval_loc#
→ incomplete
?
eval'
eval_loc'

Generation


type exp
type exploc
disornamentation
eval
eval_loc
ornamentation
eval_loc#
→ incomplete
comparison to
find the patches p
eval_locN
→ normalized version
?
eval'
eval_loc'

Generation


type exp
type exploc
disornamentation
eval
eval_loc
ornamentation
with patches p
eval_loc#
→ incomplete
comparison to
find the patches p
eval_locN
→ normalized version
?
eval'
eval_loc'
ornamentation
with patches p

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
- OCaml implementation: ongoing work for ornamentation

- Examples:
  http://gallium.inria.fr/~remy/ornaments/disorn/

- Try the online prototype:
  https://www.eleves.ens.fr/home/lbaudin/demo