Reification by Parametricity

Fast Setup for Proof by Reflection, in Two Lines of Ltac

ITP 2018

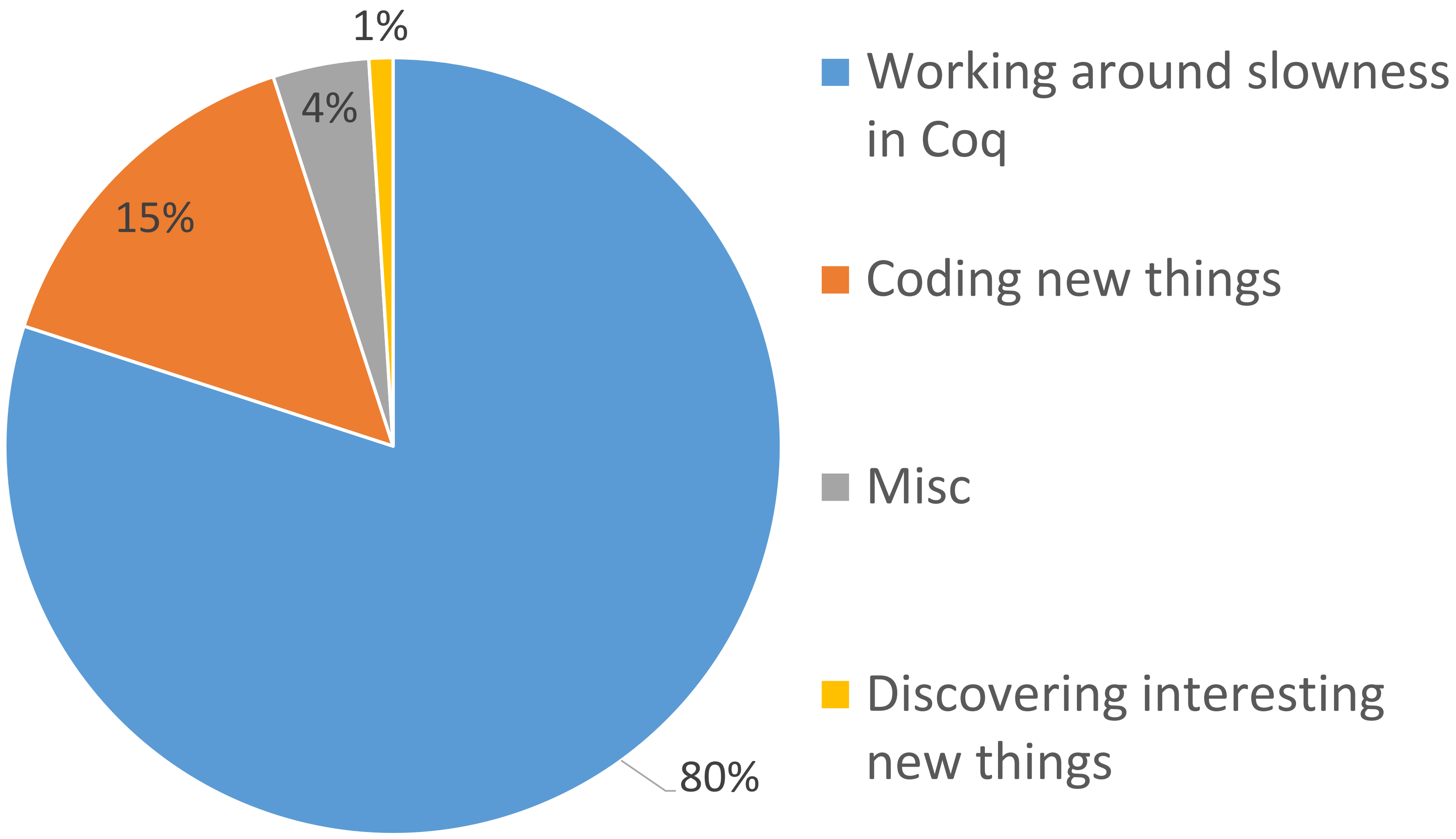
Jason Gross, Andres Erbsen, and Adam Chlipala

Reification

a technique for making proofs check faster

(and also more predictably)

Time Spent in my PhD



Reification by Parametricity

or

“A solution to

‘my technique for making my proofs check faster is too slow’.”

Outline

- Introduction
  - What is proof?
  - When is proof slow?
  - What is proof by reflection?
  - What is reification?
  - When is reification slow or complicated?
- Reification by parametricity
  - What is it?
  - What's special about it?
- What's left?

What is proof?

```
Inductive is_even : nat → Prop :=
|even_O :is_even O
|even_SS:∀x,is_even x → is_even (S (S x)).

Theorem is_even_two : is_even 2.
Proof. repeat constructor. Qed.

Print is_even_two.
(* is_even_two = even_SS 0 even_O *)
```

When is proof slow?

```
Goal is_even 9002.
  Time repeat constructor.
    (* 55.966 secs *)
Qed.
Goal is_even
  (let x := 100 * 100 * 100 * 100 in
   let y := x * x * x * x in
   y * y * y * y).
 cbv. (* stack overflow *)
Abort.
```

Why is proof slow?

```
Goal is_even 9002.
  Time repeat constructor.
    (* 55.966 secs *)

  Show Proof.
  (*even_SS 9000 (even_SS 8998 … )*)
  Set Printing All. Check 9002.
  (*S (S (S (S (S (S (S … ))))))):nat*)
```

What is proof by reflection?

```
Inductive is_even      nat → Prop :=
|even_O :is_eve
|even_SS:∀x,is_  ven     → is_even (S (S x)).

Fixpoint check_is_even (n : nat) : bool
  := match n with
     | O ⇒ true
     | S n′ ⇒ ¬ check_is_even n′
     end.
```

What is proof by reflection?

```
Theorem soundness
 : ∀n, check_is_even n = true → is_even n.

Goal is_even 9002.
 Time repeat constructor.(*55.966 s *)
 Undo.
 Time apply soundness; vm_compute;
   reflexivity. (* 0.035 s *)
```

What is proof by reflection?

```
Theorem soundness
 : ∀n, check_is_even n = true → is_even n.

Goal is_even 9002.
 Time apply soundness; vm_compute;
   reflexivity. (* 0.035 secs *)
 Show Proof.
 (* soundness 9002 eq_refl *)
```

What is proof by reflection?

```
Theorem soundness
 : ∀n, check_is_even n = true → is_even n.

Goal is_even
       (10*10*10*10*10*10*10*10*10).
 Time apply soundness; vm_compute;
        reflexivity.
 (* 174.322 secs *)
```

What is reification?

```
Fixpoint check_is_even (n : nat) : bool
  := match n with
     | O ⇒ true
     | S n′ ⇒ ¬ check_is_even n′
     end.
Fixpoint check_is_even (n : expr)
     : bool
  := match n with
     | NatO ⇒ true
     | NatS n′ ⇒ ¬ check_is_even n′
     | NatMul x y ⇒
        check_is_even x || check_is_even y
     end.
```

What is reification?

```
Inductive expr :=
| NatO : expr
| NatS : expr → expr
| NatMul : expr → exp

Reify : nat → expr
Reify O       := NatO
Reify (S n)   := NatS (Reify n)
Reify (x*y)   := NatMul (Reify x) (Reify y)
```

Requires metaprogramming!

What is reification?

Example in Ltac:

```
Ltac reify term :=
  lazymatch term with
  | O => NatO
  | S ?x => let rx := reify x in
            constr:(NatS rx)
  | ?x * ?y => let rx := reify x in
               let ry := reify y in
               constr:(NatMul rx ry)
  end.
```

When is reification complicated?

When binders show up

```
Inductive expr {var : Set} :=
| NatO : expr
| NatS : expr → expr
| NatMul : expr → expr → expr
| Var : var → expr
| LetIn : expr → (var → expr) → expr
.
```

When is reification complicated?

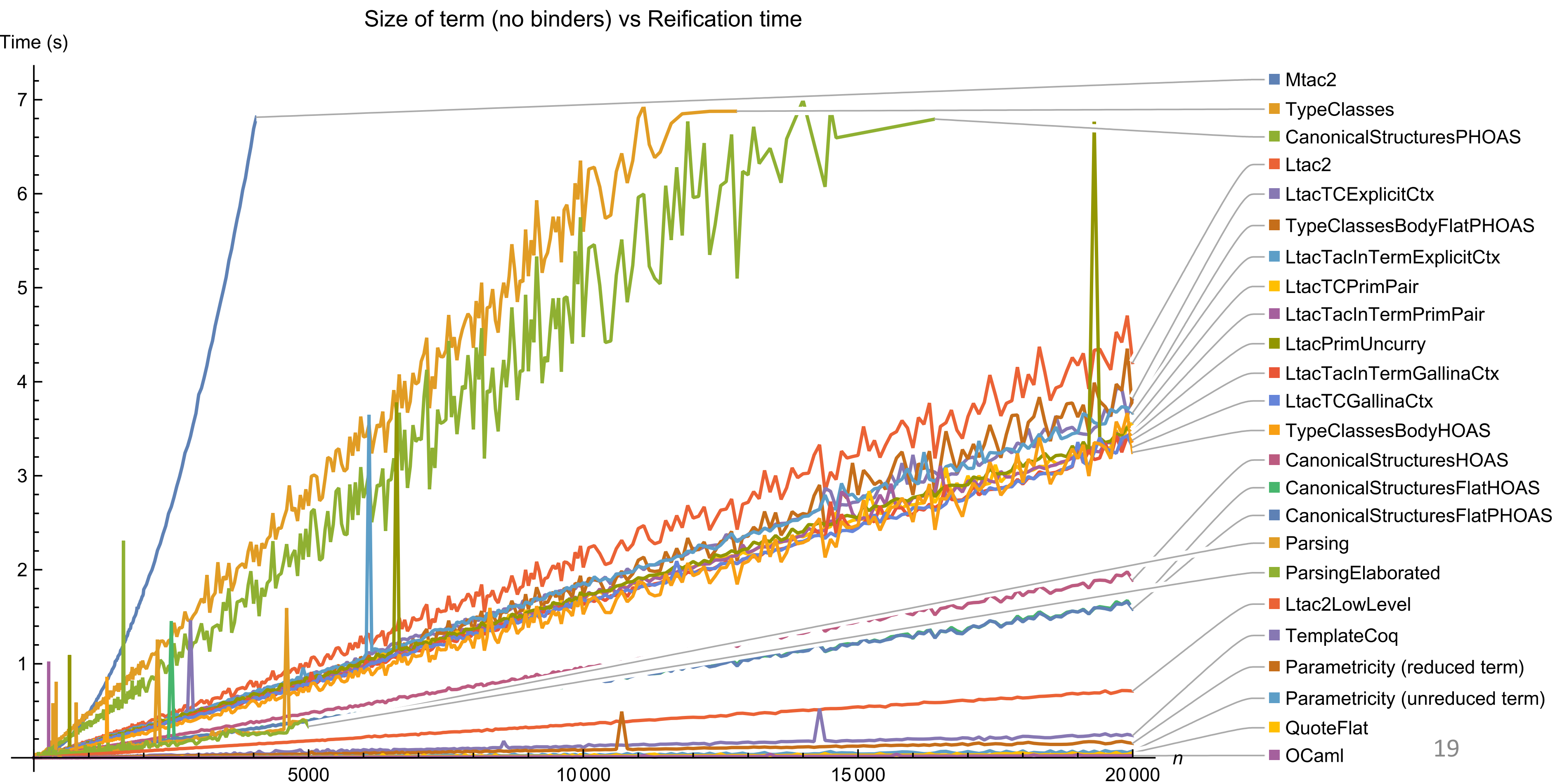
```
Reify : nat → expr
Reify O      := NatO
Reify (S n)  := NatS (Reify n)
Reify (x*y)  := NatMul (Reify x) (Reify y)
Reify (let x := v in f)
  := LetIn (Reify v)
           (λ x : var, Reify f)
```

Ltac alone admits seven(!) variants of recursing under binders.

When is reification slow?

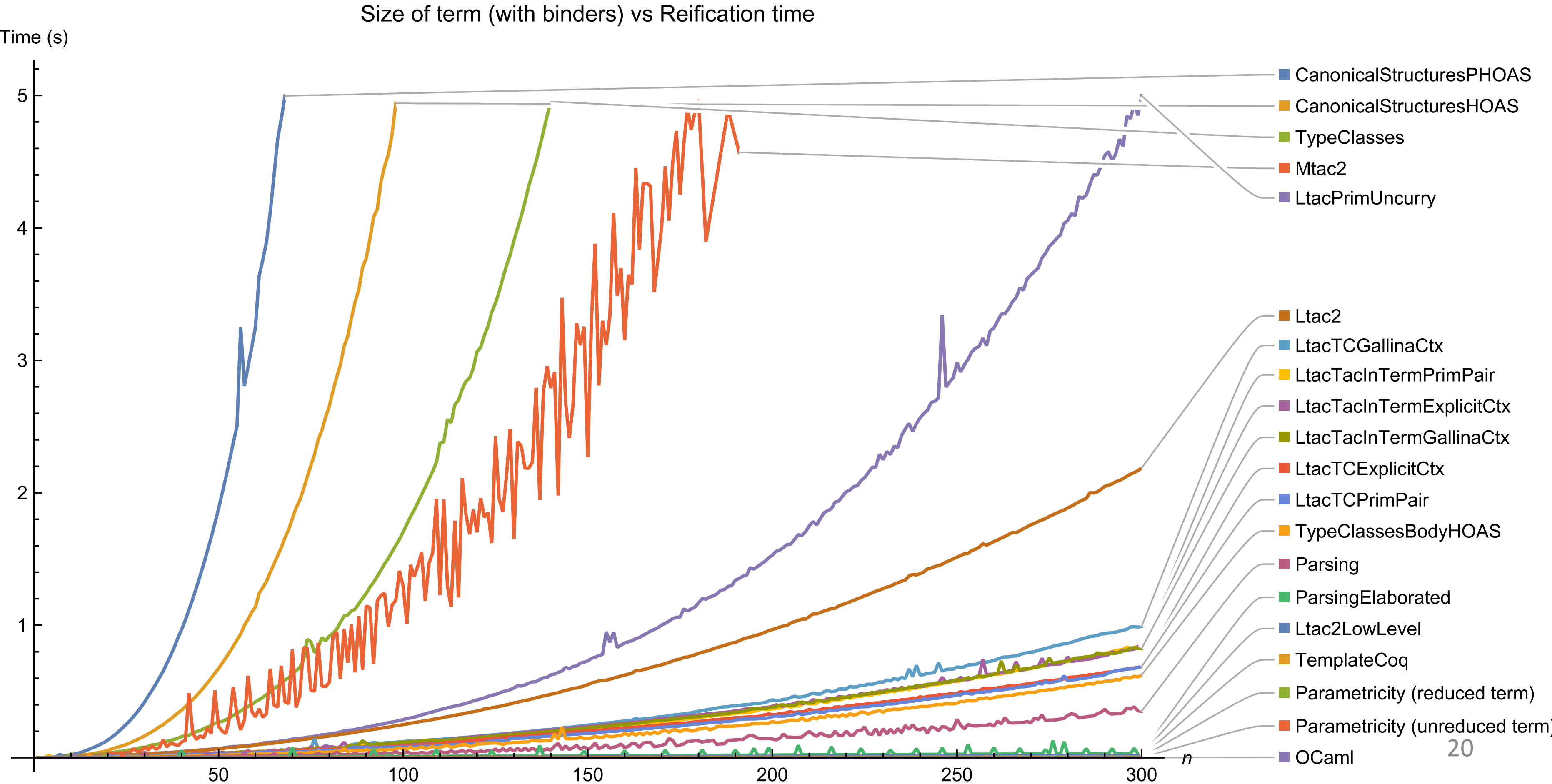
When is reification slow?

On big terms



When is reification slow?

On big terms with many binders



Reification by Parametricity

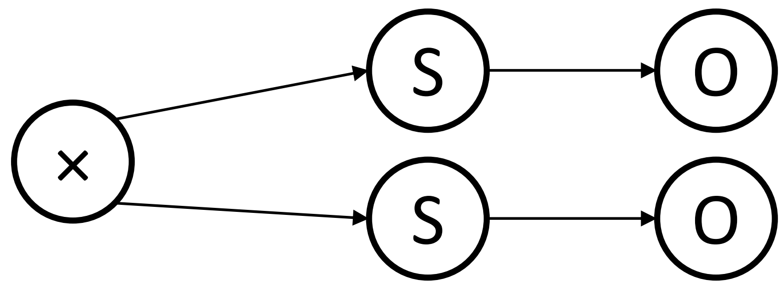
What is reification by parametricity?

Key idea:

The initial and reified terms have the *same shape.*

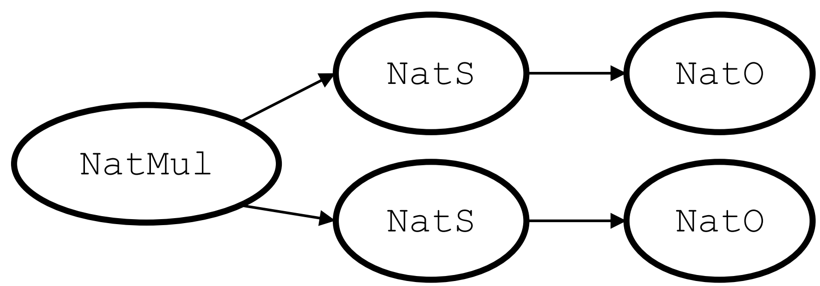
Initial term:

1×1 = Nat.mul (S O) (S O)



Reified term:

```
NatMul (NatS NatO)
       (NatS NatO)
```

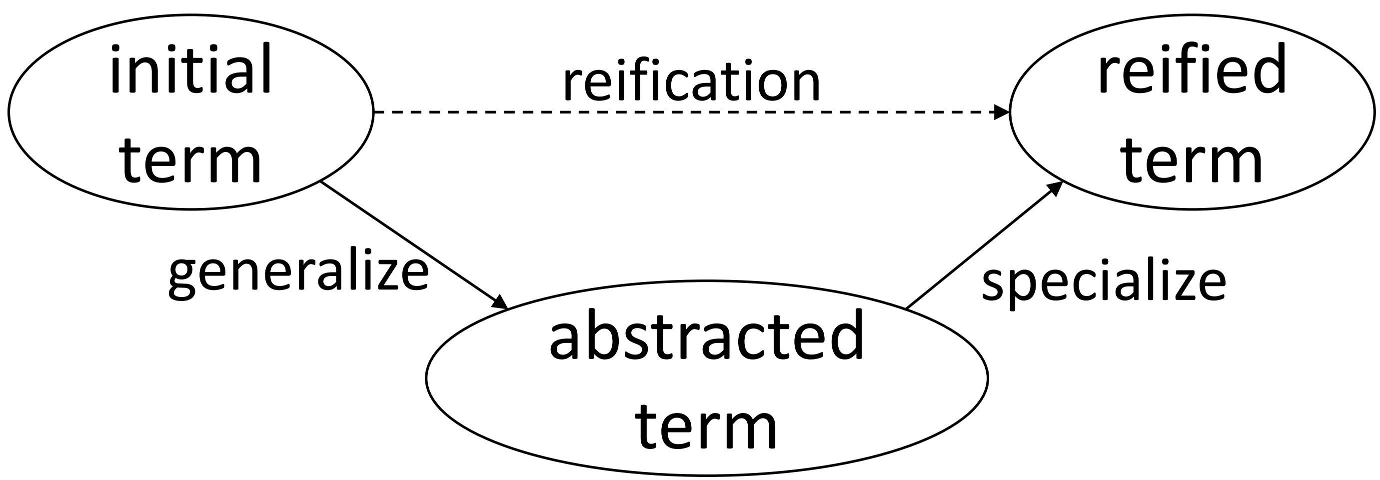


What is reification by parametricity?

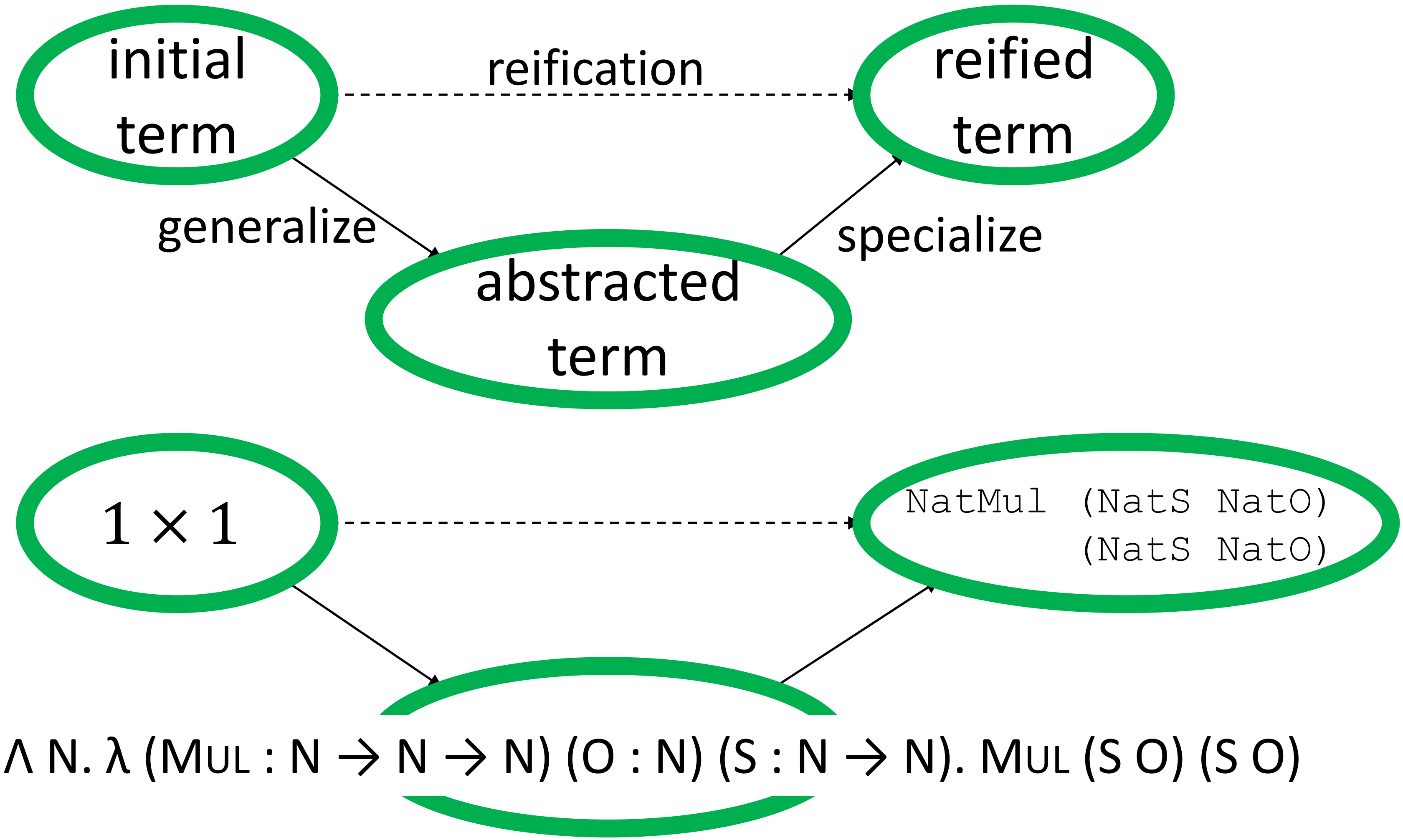
Key idea:

The initial and reified terms have the *same shape.*

We can *abstract* or *generalize* to get this shape,
and *specialize* or *substitute* to reify.



What is reification by parametricity?



Reification by Parametricity: What’s special about it?

- Concise
- Fast
- Powerful

Reification by Parametricity: It’s Concise

OCaml Reification:

```
(*i camlp4deps: "parsing/grammar.cma" i*)
(*i camlp4use: "pa_extend.cmp" i*)

open Names

let rec unsafe_reify_helper
        (mkVar : Constr.t -> 'a)
        (mkO : 'a)
        (mkS : 'a -> 'a)
        (mkOp : 'a -> 'a -> 'a)
        (mkLetIn : 'a -> Name.t -> Constr.t -> 'a -> 'a)
        (gO : Constr.t)
        (gS : Constr.t)
        (gOp : Constr.t)
        (gLetIn : Constr.t)
        (unrecognized : Constr.t -> 'a)
        (term : Constr.t)
    =
      let reify_rec term =
        unsafe_reify_helper
          mkVar mkO mkS mkOp mkLetIn gO gS gOp gLetIn unrecognized term in
      let kterm = Constr.kind term in
      if Constr.equal term gO
      then mkO
      else begin match kterm with
      | Term.Rel _ -> mkVar term
      | Term.Var _ -> mkVar term
      | Term.Cast (term, _, _) -> reify_rec term
      | Term.App (f, args)
        ->
        if Constr.equal f gS
        then let x = Array.get args 0 in
             let rx = reify_rec x in
             mkS rx
        else if Constr.equal f gOp
        then let x = Array.get args 0 in
             let y = Array.get args 1 in
             let rx = reify_rec x in
             let ry = reify_rec y in
             mkOp rx ry
        else if Constr.equal f gLetIn
        then let x = Array.get args 2 (* assume the first two args are type
params *) in
             let f = Array.get args 3 in
             begin match Constr.kind f with
             | Term.Lambda (idx, ty, body)
               -> let rx = reify_rec x in
                  let rf = reify_rec body in
                  mkLetIn rx idx ty rf
             | _ -> unrecognized term
             end
         else unrecognized term
      | _
        -> unrecognized term
      end

let unsafe_reify
        (cVar : Constr.t)
        (cO : Constr.t)
        (cS : Constr.t)
        (cOp : Constr.t)
        (cLetIn : Constr.t)
        (gO : Constr.t)
        (gS : Constr.t)
        (gOp : Constr.t)
        (gLetIn : Constr.t)
        (var : Constr.t)
        (term : Constr.t) : Constr.t =
    let mkApp0 (f : Constr.t) =
        Constr.mkApp (f, [| var |]) in
    let mkApp1 (f : Constr.t) (x : Constr.t) =
        Constr.mkApp (f, [| var ; x |]) in
    let mkApp2 (f : Constr.t) (x : Constr.t) (y : Constr.t) =
        Constr.mkApp (f, [| var ; x ; y |]) in
    let mkVar (v : Constr.t) = mkApp1 cVar v in
    let mkO = mkApp0 cO in
    let mkS (v : Constr.t) = mkApp1 cS v in
    let mkOp (x : Constr.t) (y : Constr.t) = mkApp2 cOp x y in
    let mkcLetIn (x : Constr.t) (y : Constr.t) = mkApp2 cLetIn x y in
    let mkLetIn (x : Constr.t) (idx : Name.t) (ty : Constr.t) (fbody :
Constr.t)
        = mkcLetIn x (Constr.mkLambda (idx, var, fbody)) in
    let ret = unsafe_reify_helper
              mkVar mkO mkS mkOp mkLetIn gO gS gOp gLetIn
              (fun term -> term)
              term in
    ret

let unsafe_Reify
        (cVar : Constr.t)
        (cO : Constr.t)
        (cS : Constr.t)
        (cOp : Constr.t)
        (cLetIn : Constr.t)
        (gO : Constr.t)
        (gS : Constr.t)
        (gOp : Constr.t)
        (gLetIn : Constr.t)
        (idvar : Id.t)
        (varty : Constr.t)
        (term : Constr.t) : Constr.t =
    let fresh_set = let rec fold accu c = match Constr.kind c with
      | Constr.Var id -> Id.Set.add id accu
      | _ -> Constr.fold fold accu c
      in
      fold Id.Set.empty term in
    let idvar = Namegen.next_ident_away_from
                  idvar
                  (fun id -> Id.Set.mem id fresh_set) in
    let var = Constr.mkVar idvar in
    let rterm = unsafe_reify cVar cO cS cOp cLetIn gO gS gOp gLetIn var term
in
    let rterm = Vars.substn_vars 1 [idvar] rterm in
    Constr.mkLambda (Name.Name idvar, varty, rterm)

DECLARE PLUGIN "reify"

open Ltac_plugin

open Stdarg
open Tacarg
open Names

(** Stolen from plugins/setoid_ring/newring.ml *)
open Tacexpr
open Misctypes
open Tacinterp
(* Calling a locally bound tactic *)
let ltac_lcall tac args =
  TacArg(Loc.tag @@ TacCall (Loc.tag (ArgVar(Loc.tag @@ Id.of_string
tac),args)))

let ltac_apply (f : Value.t) (args: Tacinterp.Value.t list) =
  let fold arg (i, vars, lfun) =
    let id = Id.of_string ("x" ^ string_of_int i) in
    let x = Reference (ArgVar (Loc.tag id)) in
    (succ i, x :: vars, Id.Map.add id arg lfun)
  in
  let (_, args, lfun) = List.fold_right fold args (0, [], Id.Map.empty) in
  let lfun = Id.Map.add (Id.of_string "F") f lfun in
  let ist = { (Tacinterp.default_ist ()) with Tacinterp.lfun = lfun; } in
  Tacinterp.eval_tactic_ist ist (ltac_lcall "F" args)

let to_ltac_val c = Tacinterp.Value.of_constr c

open Pp

TACTIC EXTEND quote_term_cps
    | [ "quote_term_cps" "[" ident(idvar) "," constr(varty) "]"
          constr(cVar) constr(cO) constr(cS) constr(cOp) constr(cLetIn)
          constr(gO) constr(gS) constr(gOp) constr(gLetIn)
          constr(term) tactic(tac) ] ->
      [ (** quote the given term, pass the result to t **)
  Proofview.Goal.enter begin fun gl ->
          let _ (*env*) = Proofview.Goal.env gl in
          let c = unsafe_Reify
                   (EConstr.Unsafe.to_constr cVar)
                   (EConstr.Unsafe.to_constr cO)
                   (EConstr.Unsafe.to_constr cS)
                   (EConstr.Unsafe.to_constr cOp)
                   (EConstr.Unsafe.to_constr cLetIn)
                   (EConstr.Unsafe.to_constr gO)
                   (EConstr.Unsafe.to_constr gS)
                   (EConstr.Unsafe.to_constr gOp)
                   (EConstr.Unsafe.to_constr gLetIn)
                   idvar
                   (EConstr.Unsafe.to_constr varty)
                   (EConstr.Unsafe.to_constr term) in
          ltac_apply tac (List.map to_ltac_val [EConstr.of_constr c])
  end ]
END;;
```

Reification by Parametricity: It’s Concise

Ltac Reification:

```
Definition var_for {var : Type} (n : nat) (v : var) := False.
Ltac reify var term :=
  let reify_rec term := reify var term in
  lazymatch goal with
  | [ H : var_for term ?v |- _ ] => constr:(@Var var v)
  | _ =>
    lazymatch term with
    | O => constr:(@NatO var)
    | S ?x => let rx := reify_rec x in constr:(@NatS var rx)
    | ?x * ?y => let rx := reify_rec x in let ry := reify_rec y in constr:(@NatMul var rx ry)
    | (dlet x := ?v in ?f)
      => let rv := reify_rec v in
         let not_x := fresh in
         let not_x2 := fresh in
         let rf := lazymatch constr:(
                     fun (x : nat) (not_x : var) (_ : @var_for var x not_x)
                     => match f return @expr var with
                        | not_x2
                          => ltac:(let fx := (eval cbv delta [not_x2] in not_x2) in
                                   clear not_x2;
                                   let rf := reify_rec fx in
                                   exact rf)
                        end) with
              | fun _ v' _ => @?f v' => f
              | ?f => error_cant_elim_deps f
              end in
         constr:(@LetIn var rv rf)
    | ?v => error_bad_term v
    end
  end.
```

Reification by Parametricity: It’s Concise

Typeclass-based Reification:

```
Local Generalizable Variables x y rx ry f rf.
Section with_var.
  Context {var : Type}.

  Class reify_of (term : nat) (rterm : @expr var) := {}.
  Global Instance reify_NatMul `{reify_of x rx, reify_of y ry}
    : reify_of (x * y) (rx * ry).
  Global Instance reify_LetIn `{reify_of x rx}
         `{forall y ry, reify_of y (Var ry) -> reify_of (f y) (rf ry)}
    : reify_of (dlet y := x in f y) (elet ry := rx in rf ry).
  Global Instance reify_S `{reify_of x rx}
    : reify_of (S x) (NatS rx).
  Global Instance reify_O
    : reify_of O NatO.
End with_var.
Ltac Reify x :=
  let c := constr:(fun var => (_ : @reify_of var x _)) in
  lazymatch type of c with
  | forall var, reify_of _ (@?rx var) => rx
  end.
```

Reification by Parametricity: It’s Concise

Reification by Parametricity:

```
Ltac reify var x :=
match(eval pattern nat, O, S, Nat.mul in x)with ?rx _ _ _ _ ⇒
constr:(rx (@expr var) NatO NatS NatMul) end.
```

Reification by Parametricity: It’s Concise

Reification by Parametricity (with binders):

```
Ltac reify var x :=
match(eval pattern nat, O, S, Nat.mul, (@Let_In nat nat) in x)with ?rx _ _ _ _ _ ⇒
constr:(rx (@expr var) NatO NatS NatMul (λ x' f', LetIn x' (λ v, f' (Var v)))) end.
```

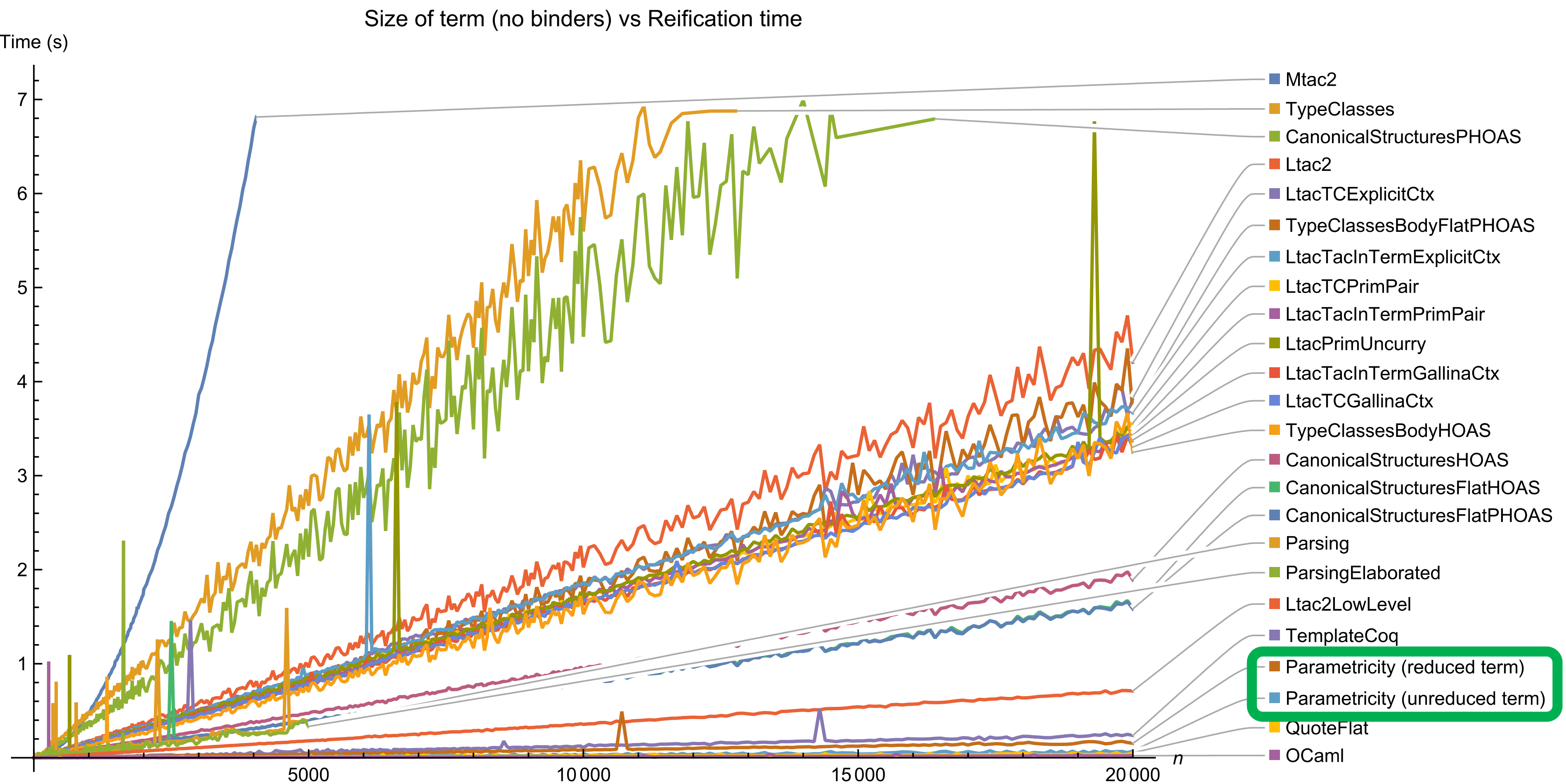
Reification by Parametricity: It’s Concise

Reification by Parametricity:

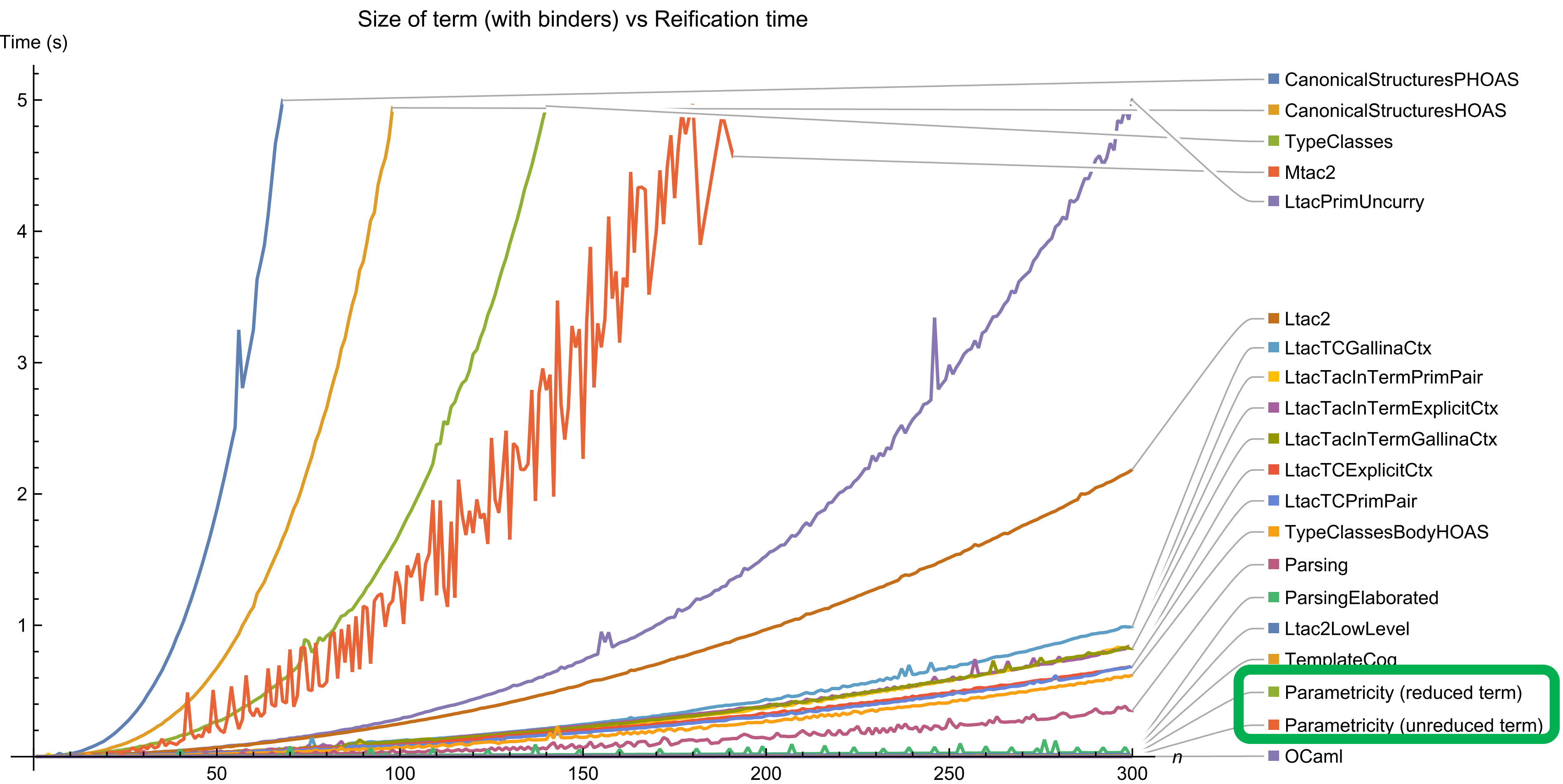
```
1. let x := constr:(1 * 1) in
2. let x := (eval pattern nat, O, S, Nat.mul in x) in
3. let x := match x with ?rx _ _ _ _ ⇒ rx end in
4. let x := constr:(x (@expr var) NatO NatS NatMul) in
5. let x := (eval cbv beta in x) in
   x
```

```
1. x = 1 * 1
2. x = ((λ N o s m, m (s o) (s o)) nat 0 S Nat.mul)
3. x =  (λ N o s m, m (s o) (s o))
4. x = ((λ N o s m, m (s o) (s o)) expr NatO NatS NatMul)
5. x = NatMul (NatS NatO) (NatS NatO)
```

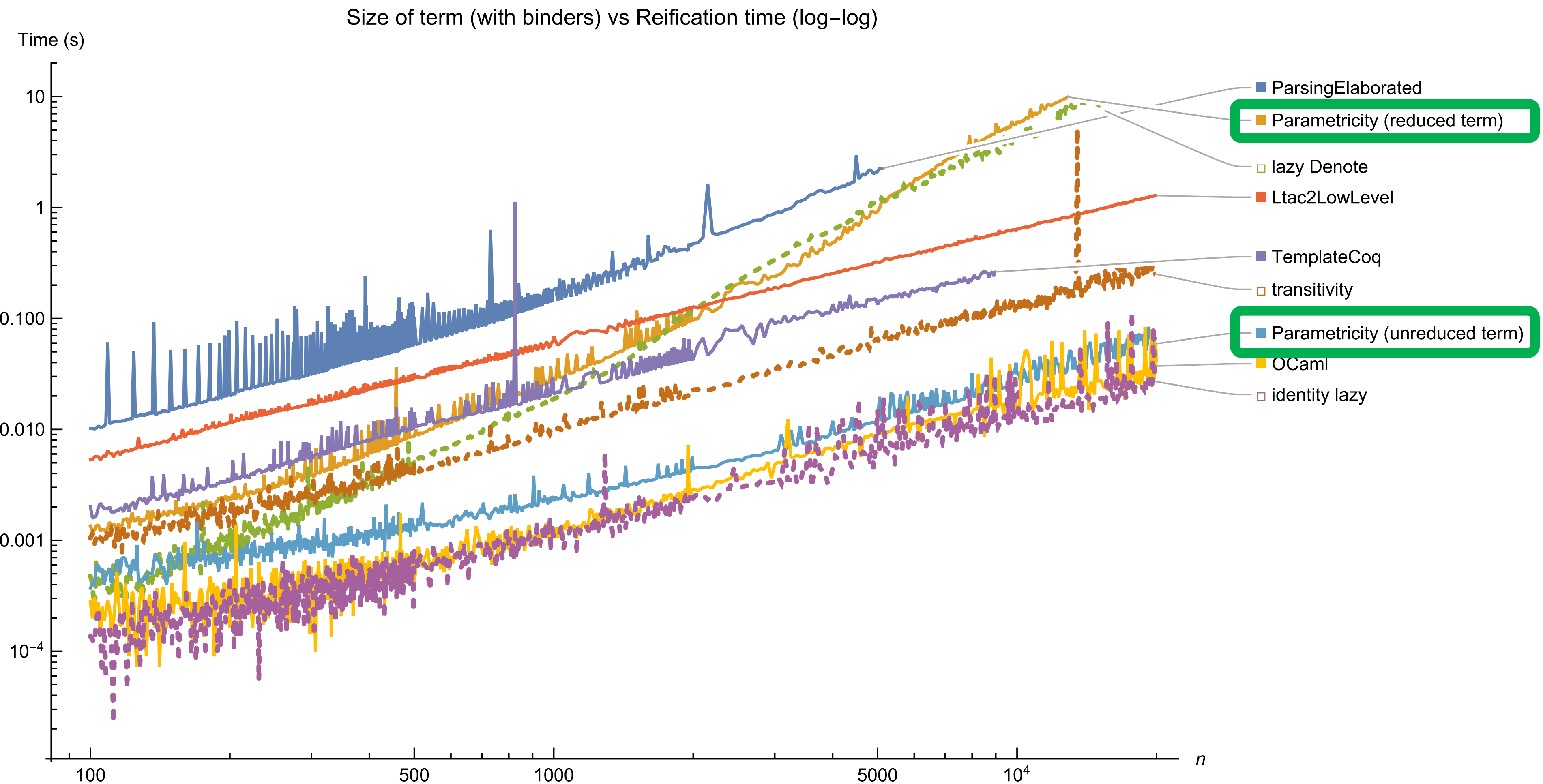
Reification by Parametricity: It's Fast



Reification by Parametricity: It's Fast



Reification by Parametricity: It's Fast (TODO: Make 8.8 Graph)



Reification by Parametricity: It's Powerful

We can *commute* reduction and reification.

Reification by Parametricity: It's Powerful

$\mathrm{dlet}\, x_1 \coloneqq 1 \times 1 \;\mathrm{in}$
$\mathrm{dlet}\, x_2 \coloneqq x_1 \times x_1 \;\mathrm{in}$
$\mathrm{dlet}\, x_3 \coloneqq x_2 \times x_2 \;\mathrm{in}$

…
$\mathrm{dlet}\, x_{100} \coloneqq x_{99} \times x_{99} \;\mathrm{in}$
x_{100}

Reification by Parametricity: It's Powerful

```
Inductive count :=
| none | one_more (how_many : count).

Fixpoint big (x:nat) (n:count) : nat
  := match n with
     | none => x
     | one_more n'
      => dlet x' := x * x in
         big x' n'
     end.

big 1 100
```

Reification by Parametricity: It's Powerful

Rather than reifying

$\text{dlet } x_1 \coloneqq 1 \times 1 \text{ in}$
$\text{dlet } x_2 \coloneqq x_1 \times x_1 \text{ in}$
$\text{dlet } x_3 \coloneqq x_2 \times x_2 \text{ in}$
…
$\text{dlet } x_{100} \coloneqq x_{99} \times x_{99} \text{ in}$
x_{100}

We can instead reify:

$(\lambda\, (x :\ \mathbb{N})\ (n : \text{count}).$
$\quad \text{count\_rec}\ (\mathbb{N} \to \mathbb{N})\ (\lambda x. x)\ (\lambda n'\ \ \text{big}_{n'}\ x.$
$\qquad \text{dlet } x' \coloneqq x \times x \text{ in big}_{n'}\ x'))\ 1\ 100$

Reification by Parametricity: It's Powerful

Initial term:

$$\text{count\_rec}\ (\mathbb{N} \to \mathbb{N})\ (\lambda x.\, x)\ (\lambda n'\ \ \text{big}_{n'}\ x.$$
$$\text{dlet}\ x' \coloneqq x \times x\ \text{in}\ \text{big}_{n'}\ x')\ 100\ 1$$

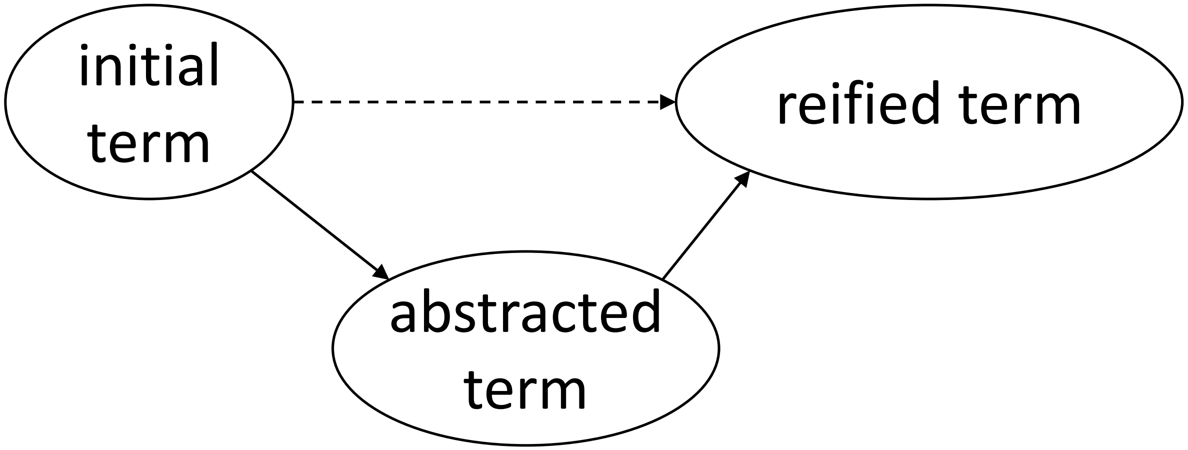
Abstracted term:

$$\Lambda\, \mathrm{N}.\, \lambda\, \textsc{Mul}\ \ \mathrm{O}\ \ \mathrm{S}\ \ \textsc{LetIn}.$$
$$\text{count\_rec}\ (\mathrm{N} \to \mathrm{N})\ (\lambda x.\, x)\ (\lambda n'\ \ \text{big}_{n'}\ x.$$
$$\textsc{LetIn}\ \ (\textsc{Mul}\ x\ x)\ (\lambda x'.\, \text{big}_{n'}\ x'))\ 100\ (\mathrm{S}\ \mathrm{O})$$

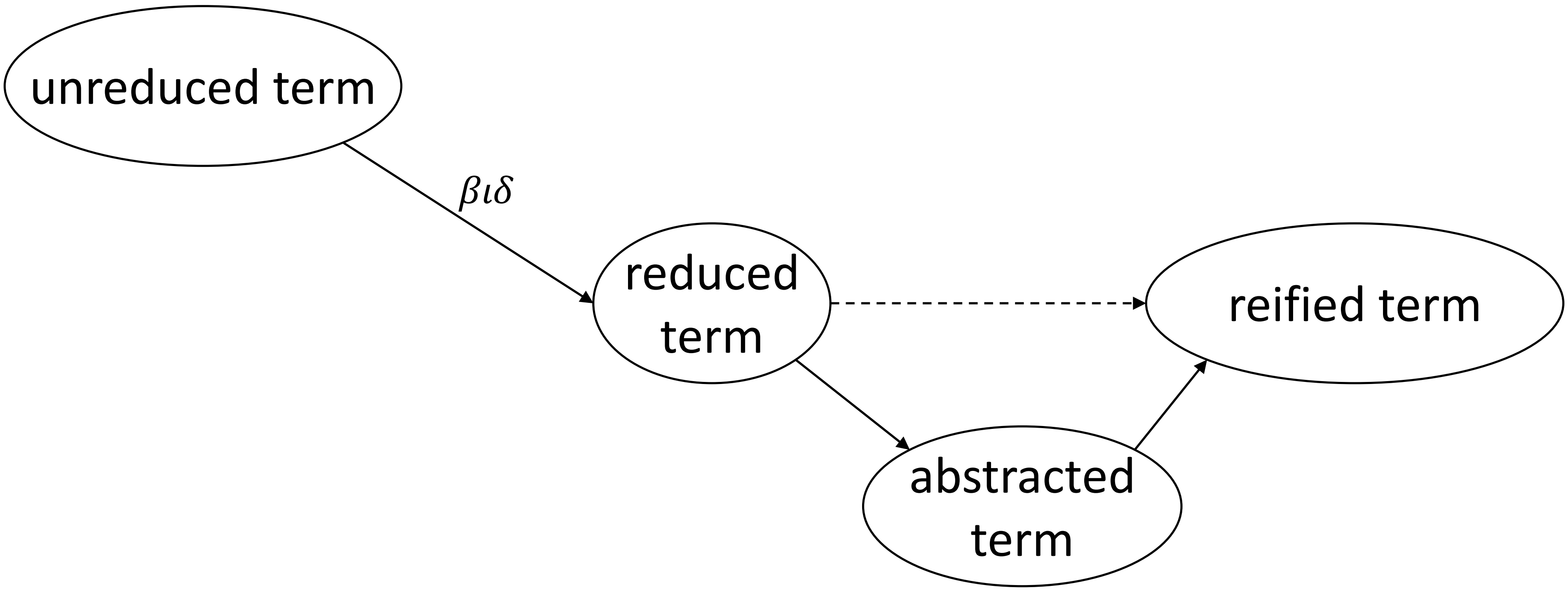
Reified term:

$$\text{count\_rec}\ (\text{expr} \to \text{expr})\ (\lambda x.\, x)$$
$$(\lambda n'\ \text{big}_{n'}\ x.\, \text{LetIn}\ (\text{NatMul}\ x\ x)\ (\lambda x'.\, \text{big}_{n'} (\text{Var}\ x')))$$
$$100\ (\text{NatS}\ \text{NatO})$$

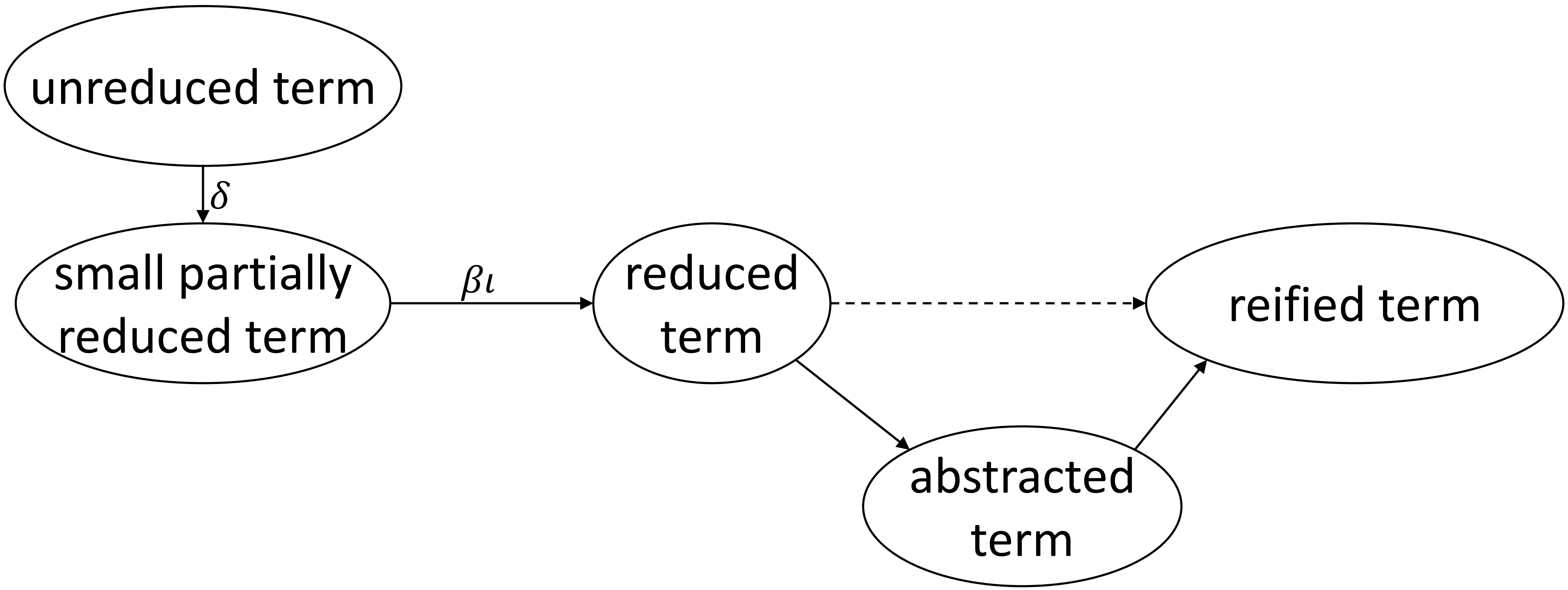
Reification by Parametricity: It's Powerful



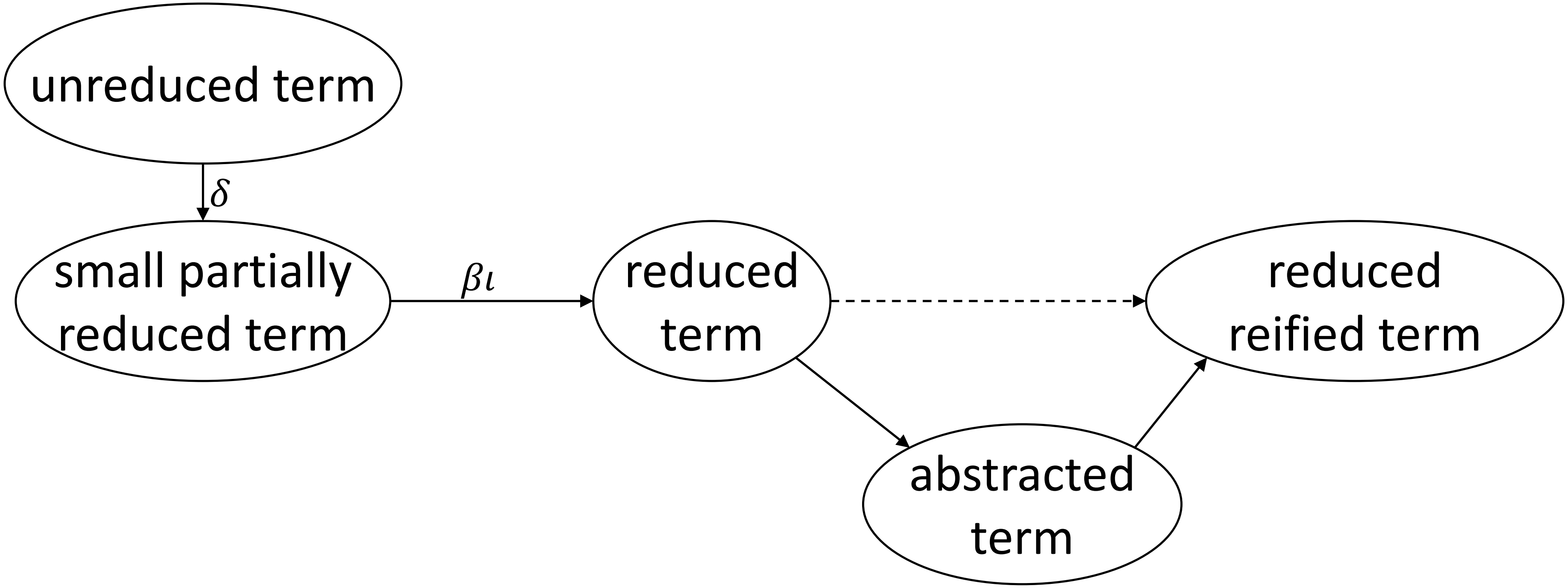
Reification by Parametricity: It's Powerful



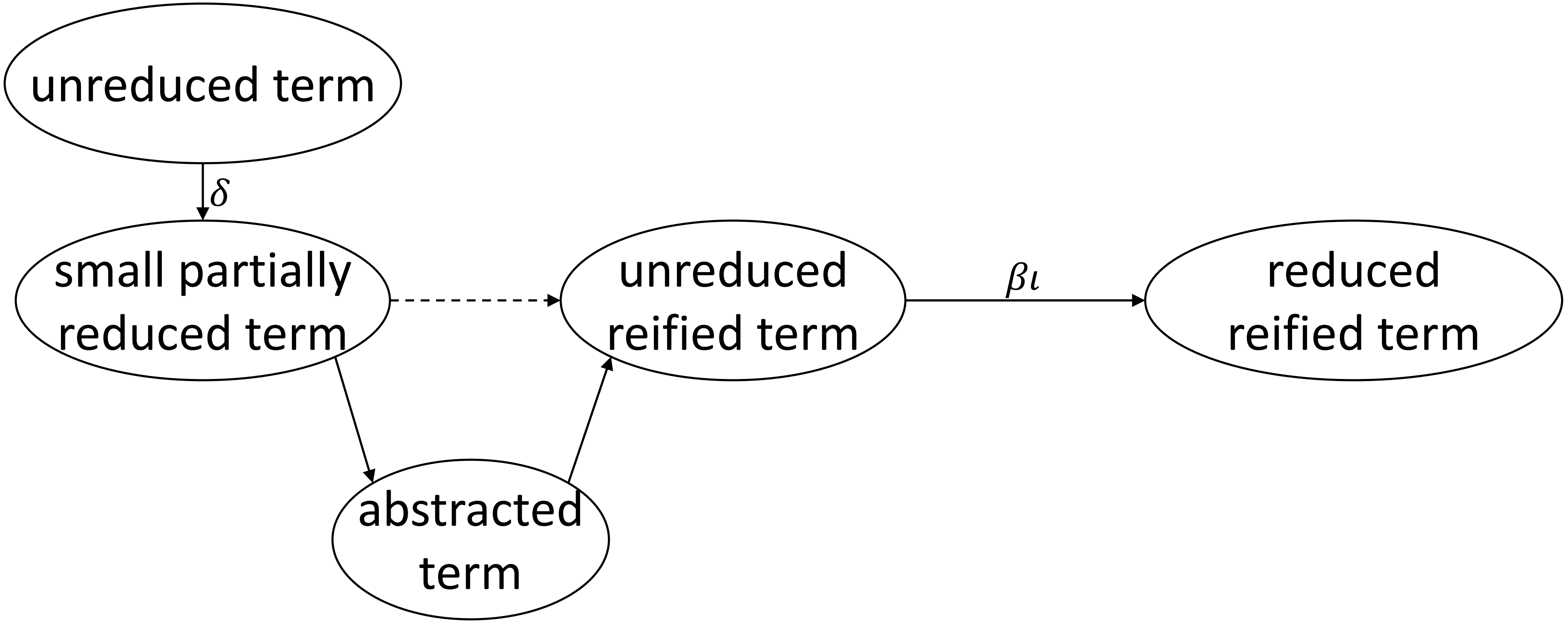
Reification by Parametricity: It's Powerful



Reification by Parametricity: It's Powerful



Reification by Parametricity: It's Powerful



What's left?

- Nuances of handling language primitives
  - $\forall/\Pi/\rightarrow$, let … in …, match/fix – handled by wrapping
  - Top-level λ – ad-hoc handling
  - Non top-level λ – handled nearly automatically
  - See paper or ask me for details
- Commuting $\beta\iota$ reduction with denotation-correctness proof
  - Seems to require parametricity
  - Future work!

Takeaways (if things went well)

- Reification is useful for making proofs check faster
- Reification by parametricity is
  - based on the insight that reification preserves shape
  - concise
  - powerful (can commute reduction and reification)
  - fast

Thank you
Any questions?

Reification and benchmarking code and data available at
https://github.com/mit-plv/reification-by-parametricity

Paper available at http://adam.chlipala.net/papers/ReificationITP18/