Reification by Parametricity

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

ITP 2018
Reification

a technique for making proofs check faster
(and also more predictably)
Time Spent in my PhD

- 80% Working around slowness in Coq
- 15% Coding new things
- 4% Misc
- 1% Discovering interesting new things
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 0
  | 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 * x in
    y * y * y * y).
cbv. (* stack overflow *)
Abort.
Why is proof slow?

Goal \textit{is\_even} 9002.

\textbf{Time repeat constructor.}
(* 55.966 secs *)

\textbf{Show Proof.}
(*\textit{even\_SS} 9000 (\textit{even\_SS} 8998 ... )* )

\textbf{Set Printing All. Check 9002.}
(*\textit{S (S (S (S (S (S ... ))))):nat}* )
What is proof by reflection?

```
Inductive is_even : nat → Prop :=
| even_0 : is_even 0 |
| even_SS : ∀ x, is_even x → is_even (S (S x)).

Fixpoint check_is_even (n : nat) : bool :=
  match n with
  | 0 ⇒ true
  | S n' ⇒ ¬ check_is_even n'
end.
```
What is proof by reflection?

**Theorem soundness**

\[ \forall n, \text{check\_is\_even}\ n = \text{true} \rightarrow \text{is\_even}\ n. \]

**Goal** is\_even 9002.

**Time** repeat\_constructor. (*55.966 s *)

Undo.

**Time** apply soundness; vm\_compute; \text{reflexivity}. (* 0.035 s *)
What is proof by reflection?

Theorem soundness

: \( \forall n, \ \text{check\_is\_even} \ n = \text{true} \rightarrow \text{is\_even} \ n. \)

Goal \( \text{is\_even} \ 9002. \)

Time apply soundness; vm_compute; reflexivity. (* 0.035 secs *)

Show Proof.

(* soundness 9002 eq\_refl *)
What is proof by reflection?

**Theorem soundness**

: \( \forall n, \text{check_is_even } n = \text{true } \rightarrow \text{is_even } n. \)

**Goal is_even**

(10*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 → expr → expr

**Reify : nat → expr**

- **Reify 0** := 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

```ml
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?

\textbf{Reify} : \texttt{nat} $\rightarrow$ \texttt{expr}

\textbf{Reify} 0 := \texttt{NatO}

\textbf{Reify} (S n) := \texttt{NatS} (\text{Reify} n)

\textbf{Reify} (x*y) := \texttt{NatMul} (\text{Reify} x) (\text{Reify} y)

\textbf{Reify} (let x := v in f) := \texttt{LetIn} (\text{Reify} v)

\hspace{1cm} (\lambda x : \texttt{var}, \text{Reify} f)

Ltac alone admits seven(!) variants of recursing under binders.
When is reification slow?
When is reification slow?

On big terms

Size of term (no binders) vs Reification time
When is reification slow?

On big terms with many binders

Size of term (with binders) vs Reification time
Reification by Parametricity
What is reification by parametricity?

**Key idea:**
The initial and reified terms have the *same shape*.

**Initial term:**
\[ 1 \times 1 = \text{Nat.mul} (S \ O) (S \ O) \]

**Reified term:**
\[ \text{NatMul} \ (\text{NatS} \ \text{NatO}) \]
\[ (\text{NatS} \ \text{NatO}) \]
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?

- **Initial term**
- **Abstracted term**
- **Reified term**

**Reification**

**Generalize**

**Specialize**

**1 × 1**

\[ \Lambda N. \lambda (MUL : N \to N \to N) (O : N) (S : N \to N). MUL (S O) (S O) \]

NatMul (NatS NatO) (NatS NatO)
Reification by Parametricity: What’s special about it?

• Concise

• Fast

• Powerful
Reification by Parametricity: It’s Concise

OCaml Reification:
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 0 Nat0.
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

Size of term (no binders) vs Reification time
Reification by Parametricity: It’s Fast

Size of term (with binders) vs Reification time

- CanonicalStructuresPHOAS
- CanonicalStructuresHOAS
- TypeClasses
- Ltac2
- LtacPrimUncurry
- Ltac2
- LtacTGGallinaCtx
- LtacTacInTermPrimPair
- LtacTacInTermExplicitCtx
- LtacTacInTermGallinaCtx
- LtacTCExplicitCtx
- LtacTCPrimPair
- TypeClassesBodyHOAS
- Parsing
- ParsingElaborated
- Ltac2LowLevel
- TemplateCog
- Parametricity (reduced term)
- Parametricity (unreduced term)
Reification by Parametricity: It’s Fast (TODO: Make 8.8 Graph)

Size of term (with binders) vs Reification time (log-log)
Reification by Parametricity: It’s Powerful

We can *commute* reduction and reification.
Reification by Parametricity: It’s Powerful

dlet \( x_1 \) := \( 1 \times 1 \) in

dlet \( x_2 \) := \( x_1 \times x_1 \) in

dlet \( x_3 \) := \( x_2 \times x_2 \) in

... 

dlet \( x_{100} \) := \( x_{99} \times x_{99} \) 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
\[
\begin{align*}
dlet x_1 & := 1 \times 1 \text{ in} \\
dlet x_2 & := x_1 \times x_1 \text{ in} \\
dlet x_3 & := x_2 \times x_2 \text{ in} \\
\vdots \\
dlet x_{100} & := x_{99} \times x_{99} \text{ in} \\
x_{100}
\end{align*}
\]
We can instead reify:
\[
(\lambda (x : \mathbb{N}) (n : \text{count}). \quad \\
\text{count\_rec (N \to N) (\lambda x. x) (\lambda n' \text{ big}_{n'} x.} \\
\quad \text{dlet } x' := x \times x \text{ in big}_{n'} x')) \mid 1 \ 100
\]
Reification by Parametricity:
It’s Powerful

Initial term:
```
count_rec (N → N) (λx. x) (λn′ bigₙ, x.
dlet x′ := x × x in bigₙ x′) 100 1
```

Abstracted term:
```
Λ N. λ M U L  O S L E T I N.
count_rec (N → N) (λx. x) (λn′ bigₙ, x.
LE T I N (M U L x x) (λx′. bigₙ x′)) 100 (S O)
```

Reified term:
```
count_rec (expr → expr) (λx. x)
(λn′ bigₙ, x. LetIn (NatMul x x) (λx′. bigₙ x′ (Var x′)))
100 (NatS NatO)
```
Reification by Parametricity: It’s Powerful

initial term → abstracted term → reified term
Reification by Parametricity: It’s Powerful

unreduced term

\[ \beta \iota \delta \]

reduced term

abstracted term

reified term
Reification by Parametricity: It’s Powerful

unreduced term

δ

small partially reduced term

β_t

reduced term

abstracted term

reified term
Reification by Parametricity: It’s Powerful

unreduced term

small partially reduced term

\(\delta\)

\(\beta_t\)

reduced term

abstracted term

reduced reified term
Reification by Parametricity: It’s Powerful

unreduced term

small partially reduced term

abstracted term

unreduced reified term

reduced reified term

\( \delta \)

\( \beta \)
What’s left?

• Nuances of handling language primitives
  • $\forall/\Pi/\to$, let ... in ..., match/fix – handled by wrapping
  • Top-level $\lambda$ – ad-hoc handling
  • Non top-level $\lambda$ – 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/