Implementing parametricity in Rocq-Elpi

Cyril Cohen [1] Vojtěch Štěpančík [2]

January 17, 2026

[1]Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP, UMR 5668

[2]Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N, UMR 6004

Parametricity translations

- Parametricity — property of a type system linked to relational interpretations

Parametricity translations

- Parametricity — property of a type system linked to relational interpretations
- This talk's POV: practical codegen tool

Parametricity translations

- Parametricity — property of a type system linked to relational interpretations
- This talk's POV: practical codegen tool
- In dependent type theory relations are first class — parametricity implementable as a translation from terms to terms

Examples

- **Data type refinement** (CoqEAL): proof-oriented and computation-oriented types, push computations to the efficient one, e.g.

$$\sum_{i=0_{\mathbb{N}}}^{5}(1000 +_{\mathbb{N}} i) = \uparrow \sum_{i=0_{\mathtt{N}}}^{5}(1000 +_{\mathtt{N}} i)$$

Examples

- **Data type refinement** (CoqEAL): proof-oriented and computation-oriented types, push computations to the efficient one, e.g.

$$\sum_{i=0_{\mathbb{N}}}^{5}(1000 +_{\mathbb{N}} i) = \uparrow \sum_{i=0_{\mathbb{N}}}^{5}(1000 +_{\mathtt{N}} i)$$

- **Proof transfer** (Trocq): transfer properties, e.g. show $3 \nmid xyz \Rightarrow x^3 +_{\mathbb{N}} y^3 \neq z^3$ by computing in $\mathbb{Z}_9$

Examples

- **Data type refinement** (CoqEAL): proof-oriented and computation-oriented types, push computations to the efficient one, e.g.

$$\sum_{i=0_{\mathbb{N}}}^{5}(1000 +_{\mathbb{N}} i) = \uparrow \sum_{i=0_{\mathbb{N}}}^{5}(1000 +_{\mathtt{N}} i)$$

- **Proof transfer** (Trocq): transfer properties, e.g. show $3 \nmid xyz \Rightarrow x^3 +_{\mathbb{N}} y^3 \neq z^3$ by computing in $\mathbb{Z}_9$
- **Displayed objects** (Rocq-Elpi): derive dependent versions of datatypes and use them in generated induction principles

Standard presentation

Translate contexts

$$[\![\cdot]\!] = \cdot \qquad\qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\ x\ x'$$

Standard presentation

Translate contexts

$$[\![\cdot]\!] = \cdot \qquad\qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\ x\ x'$$

Translate terms

$$\begin{aligned}
[\![\square_i]\!] &= \lambda A\ A' \Rightarrow A \to A' \to \square_i \\
[\![x]\!] &= x_R \\
[\![\lambda(x : A) \Rightarrow t]\!] &= \lambda(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x') \Rightarrow [\![t]\!] \\
[\![t\ u]\!] &= [\![t]\!]\ u\ u'\ [\![u]\!] \\
&\vdots
\end{aligned}$$

Standard presentation

Translate contexts

$$[\![\cdot]\!] = \cdot \qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\ x\ x'$$

Translate terms

$$\begin{aligned}
[\![\square_i]\!] &= \lambda A\ A' \Rightarrow A \to A' \to \square_i \\
[\![x]\!] &= x_R \\
[\![\lambda(x : A) \Rightarrow t]\!] &= \lambda(x : A)(x' : A')(x_R : [\![A]\!]\ x\ x') \Rightarrow [\![t]\!] \\
[\![t\ u]\!] &= [\![t]\!]\ u\ u'\ [\![u]\!] \\
&\vdots
\end{aligned}$$

But what is this $-'$ operation?

Hidden complexity

- We are running two translations — parametricity and variable duplication

Hidden complexity

- We are running two translations — parametricity and variable duplication
- We can do de Bruijn or manual state for freshness, but do we want to?

Hidden complexity

- We are running two translations — parametricity and variable duplication
- We can do de Bruijn or manual state for freshness, but do we want to?
- The problem asks us to handle two aspects: induction on syntax and binders

Hidden complexity

- We are running two translations — parametricity and variable duplication
- We can do de Bruijn or manual state for freshness, but do we want to?
- The problem asks us to handle two aspects: induction on syntax and binders
- OCaml plugins and MetaRocq can do induction, but binders are manual

Rocq-Elpi

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings

Rocq-Elpi

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

  $\lambda(x : \mathbb{B}) \Rightarrow \text{true} \wedge x$ $\quad\Leftrightarrow\quad$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

$\lambda(x : \mathbb{B}) \Rightarrow \text{true} \wedge x \quad \Leftrightarrow \quad$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

$\lambda (x : \mathbb{B}) \Rightarrow \mathrm{true} \wedge x$ $\quad\Leftrightarrow\quad$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

$\lambda(x : \mathbb{B}) \Rightarrow \text{true} \wedge x$ $\Leftrightarrow$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

$\lambda(x : \mathbb{B}) \Rightarrow \text{true} \wedge x$ $\quad\Leftrightarrow\quad$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

Inductive $W\,(A : \square) : \square := |w : A \to W.$ $\quad\Leftrightarrow$

```
parameter `A` «□» (A\ ind `W` «□» (W\
  [constr `w` (prod `_` A _\ W)]))
```

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

$\lambda(x : \mathbb{B}) \Rightarrow \mathrm{true} \wedge x$ $\quad\Leftrightarrow\quad$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

Inductive $W\,(A : \square) : \square := |w : A \to W.$ $\quad\Leftrightarrow\quad$

```
parameter `A` «□» (A\ ind `W` «□» (W\
  [constr `w` (prod `_` A _\ W)]))
```

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

$\lambda(x : \mathbb{B}) \Rightarrow \text{true} \wedge x$ $\Leftrightarrow$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

$\text{Inductive } W\ A : \square) : \square := |w : A \to W.$ $\Leftrightarrow$

```
parameter `A` «□» (A\ ind `W` «□» (W\
  [constr `w` (prod `_` A _\ W)]))
```

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

$\lambda(x : \mathbb{B}) \Rightarrow \text{true} \wedge x$ ⇔ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

Inductive $W\,(A : \square) : \square := |w : A \to W.$ ⇔

```
parameter `A` «□» (A\ ind `W` «□» (W\
  [constr `w` (prod `_` A _\ W)]))
```

- "Embeddable Lambda Prolog Interpreter" with Rocq bindings
- HOAS representation for terms and declarations

  $\lambda(x : \mathbb{B}) \Rightarrow \mathrm{true} \wedge x$ $\quad\Leftrightarrow\quad$ `fun `x` «ℕ» (x\ app [«andb» «true» x])`

  $\mathrm{Inductive}\ W\,(A : \square) : \square := |w : A \to W.$ $\quad\Leftrightarrow$

  ```
  parameter `A` «□» (A\ ind `W` «□» (W\
    [constr `w` (prod `_` A _\ W)]))
  ```

- It's right there in the name! Binders and induction

Implementation

- Switch from functional-style $[\![t]\!] = \ldots$ to sequent-style $\Xi \vdash t \sim t' \because t_R$ with t' and t_R considered outputs

Implementation

- Switch from functional-style $[\![t]\!] = \ldots$ to sequent-style $\Xi \vdash t \sim t' \because t_R$ with t' and t_R considered outputs

$$\frac{(x, x', x_R) \in \Xi}{\Xi \vdash x \sim x' \because x_R}$$

Implementation

- Switch from functional-style $[\![t]\!] = \ldots$ to sequent-style $\Xi \vdash t \sim t' \because t_R$ with t' and t_R considered outputs

$$\frac{(x, x', x_R) \in \Xi}{\Xi \vdash x \sim x' \because x_R}$$

$$\frac{}{\Xi \vdash \square_i \sim \square_i \because \lambda(A\,B : \square_i).\, A \to B \to \square_i}$$

Implementation

- Switch from functional-style $[\![t]\!] = \ldots$ to sequent-style $\Xi \vdash t \sim t' \because t_R$ with t' and t_R considered outputs

$$\frac{(x, x', x_R) \in \Xi}{\Xi \vdash x \sim x' \because x_R}$$

$$\frac{}{\Xi \vdash \square_i \sim \square_i \because \lambda(A\, B : \square_i).\, A \to B \to \square_i}$$

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash t \sim t' \because t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \because \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

Implementation

- Switch from functional-style $[\![t]\!] = \ldots$ to sequent-style $\Xi \vdash t \sim t' \because t_R$ with t' and t_R considered outputs

$$\frac{(x, x', x_R) \in \Xi}{\Xi \vdash x \sim x' \because x_R}$$

$$\frac{}{\Xi \vdash \square_i \sim \square_i \because \lambda(A\, B : \square_i).\, A \to B \to \square_i}$$

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash t \sim t' \because t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \because \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

- $t \sim t' \because t_R$ represented by $\mathrm{param}/3$

Rule implementation

- Each rule gets a clause, e.g.

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \ \because \ A_R \qquad \Xi, x \sim x' \ \because \ x_R \vdash t \sim t' \ \because \ t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \ \because \ \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

Rule implementation

- Each rule gets a clause, e.g.

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \mathrel{\because} A_R \qquad \Xi, x \sim x' \mathrel{\because} x_R \vdash t \sim t' \mathrel{\because} t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \mathrel{\because} \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

```
param(fun N A T)(fun N A' T')(fun N1 A x\ fun N2 A' x'\
   fun NR (AR x x') xR\
     TR x x' xR) :-
```

Rule implementation

- Each rule gets a clause, e.g.

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \ \because\ A_R \qquad \Xi, x \sim x' \ \because\ x_R \vdash t \sim t' \ \because\ t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \ \because\ \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

```
param(fun N A T)(fun N A' T')(fun N1 A x\ fun N2 A' x'\
    fun NR (AR x x') xR\
      TR x x' xR) :-
  derive.param2.names N N1 N2 NR,
```

Rule implementation

- Each rule gets a clause, e.g.

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \ \because\ A_R \qquad \Xi, x \sim x' \ \because\ x_R \vdash t \sim t' \ \because\ t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \ \because\ \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

```
param(fun N A T)(fun N A' T')(fun N1 A x\ fun N2 A' x'\
   fun NR (AR x x') xR\
     TR x x' xR) :-
  derive.param2.names N N1 N2 NR,
  param A A' ARt,
```

Rule implementation

- Each rule gets a clause, e.g.

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash t \sim t' \because t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \because \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

```
param(fun N A T)(fun N A' T')(fun N1 A x\ fun N2 A' x'\
   fun NR (AR x x') xR\
     TR x x' xR) :-
  derive.param2.names N N1 N2 NR,
  param A A' ARt,
  (pi x x' xR\ param x x' xR ⇒
    param (T x) (T' x') (TR x x' xR)),
```

Rule implementation

- Each rule gets a clause, e.g.

$$\frac{x, x', x_R \notin \mathrm{Var}(\Xi) \qquad \Xi \vdash A \sim A' \ \because\ A_R \qquad \Xi, x \sim x' \ \because\ x_R \vdash t \sim t' \ \because\ t_R}{\Xi \vdash \lambda x : A.\, t \sim \lambda x' : A'.\, t' \ \because\ \lambda x x' (x_R : A_R\, x\, x').\, t_R}$$

```
param(fun N A T)(fun N A' T')(fun N1 A x\ fun N2 A' x'\
    fun NR (AR x x') xR\
      TR x x' xR) :-
  derive.param2.names N N1 N2 NR,
  param A A' ARt,
  (pi x x' xR\ param x x' xR ⇒
    param (T x) (T' x') (TR x x' xR)),
  (AR = x\ x'\ {coq.mk-app ARt [x,x']})
```

Scaling from CIC to Rocq

- Rocq's unfolding of `fix` is conservative, scrutinized argument needs to be constructor-headed — insert a trivial match to make it compute

Scaling from CIC to Rocq

- Rocq's unfolding of `fix` is conservative, scrutinized argument needs to be constructor-headed — insert a trivial match to make it compute
- Records need special care if we want their translations to be records
  - Translating as inductives doesn't work, the result is an indexed inductive

```
Record W (A : □) := bld { p : A }.
```

Scaling from CIC to Rocq

- Rocq's unfolding of `fix` is conservative, scrutinized argument needs to be constructor-headed — insert a trivial match to make it compute
- Records need special care if we want their translations to be records
  - Translating as inductives doesn't work, the result is an indexed inductive

```
Record W (A : □) := bld { p : A }.                    ⇝
Inductive W (A : □) := | bld (p : A) → W.
```

Scaling from CIC to Rocq

- Rocq's unfolding of `fix` is conservative, scrutinized argument needs to be constructor-headed — insert a trivial match to make it compute
- Records need special care if we want their translations to be records
  - Translating as inductives doesn't work, the result is an indexed inductive

```
Record W (A : □) := bld { p : A }.                    ⇝
Inductive W (A : □) := | bld (p : A) → W.             ⇝
Inductive W_R (A A' A_R) : W A → W A' → □ :=
  | bld_R p p' (w_R : A_R p p') → W_R (bld p) (bld' p).
```

Scaling from CIC to Rocq

- Rocq's unfolding of `fix` is conservative, scrutinized argument needs to be constructor-headed — insert a trivial match to make it compute
- Records need special care if we want their translations to be records
  - Translating as inductives doesn't work, the result is an indexed inductive

```
Record W (A : □) := bld { p : A }.                    ⇝
Inductive W (A : □) := | bld (p : A) → W.             ⇝
Inductive W_R (A A' A_R) : W A → W A' → □ :=
  | bld_R p p' (w_R : A_R p p') → W_R (bld p) (bld' p).
```

While we want

```
Record W_R (A A' A_R) (w : W A) (w' : W A') :=
  bld_R { p_R : A_R (w.p) (w'.p) }.
```

Scaling from CIC to Rocq

- Records aren't exactly dual to inductives — inductives have the universal eliminator `match`, but records don't have a "universal constructor"

Scaling from CIC to Rocq

- Records aren't exactly dual to inductives — inductives have the universal eliminator `match`, but records don't have a "universal constructor"
- $\forall A, WA \to A \sim \forall A, WA \to A \mathrel{\dot{:}} \lambda p\, p'. \forall A\, A'\, A_R\, w\, w', W_R\, w\, w' \to A_R\, pw\, pw'$, so $W.p \sim W.p \mathrel{\dot{:}} W_R.p_R$ is okay

Scaling from CIC to Rocq

- Records aren't exactly dual to inductives — inductives have the universal eliminator `match`, but records don't have a "universal constructor"
- $\forall A, WA \to A \sim \forall A, WA \to A \ \because\ \lambda p\, p'. \forall A\, A'\, A_R\, w\, w', W_R\ w\ w' \to A_R\ pw\ pw'$, so $W.p \sim W.p \ \because\ W_R.p_R$ is okay
- But $\forall A, A \to WA \sim \forall A, A \to WA \ \because\ \lambda b\, b'. \forall A\, A'\, A_R\, a\, a', A_R\ a\ a' \to W_R\ ba\ b'\, a'$, so $\mathrm{bld} \sim \mathrm{bld} \ \because\ \mathrm{bld}_R$ is **not** okay, since $\mathrm{bld}_R : \forall A\, A'\, A_R\, \mathbf{w}\, \mathbf{w}', A_R\ \mathbf{w}.\mathbf{p}\ \mathbf{w}'.\mathbf{p} \to W_R\ \mathbf{w}\ \mathbf{w}'$

Scaling from CIC to Rocq

- Records aren't exactly dual to inductives — inductives have the universal eliminator `match`, but records don't have a "universal constructor"
- $\forall A, WA \to A \sim \forall A, WA \to A \because \lambda p\, p'. \forall A\, A'\, A_R\, w\, w', W_R\, w\, w' \to A_R\, pw\, pw'$, so $W.p \sim W.p \because W_R.p_R$ is okay
- But $\forall A, A \to WA \sim \forall A, A \to WA \because \lambda b\, b'. \forall A\, A'\, A_R\, a\, a', A_R\, a\, a' \to W_R\, ba\, b'\, a'$, so $\mathrm{bld} \sim \mathrm{bld} \because \mathrm{bld}_R$ is **not** okay, since $\mathrm{bld}_R : \forall A\, A'\, A_R\, \mathbf{w}\, \mathbf{w}', A_R\, \mathbf{w.p}\, \mathbf{w}'.\mathbf{p} \to W_R\, \mathbf{w}\, \mathbf{w}'$
- In general, constructors take arguments bundled in records, but they are expected to take them unbundled, so they need shims. Inductive eliminators would need the same if it wasn't for `match`

Conclusion

- Most of this talk exists as `derive.param2` in Rocq-Elpi
  - Records aren't upstreamed yet
- CoqEAL is using this implementation, Trocq is more complicated but uses the same ideas
- Caveat: no support for mutual inductives yet
- Give Rocq-Elpi a shot