Mechanizing Logical Relations

Josselin Poiret

Gallinette team

Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004,

F-44000 Nantes, France

June 9, 2025

Mechanization projects

Agda logrel-mltt by Abel, Öhman, and Vezzosi;

Rocq logrel-coq by Adjedj et al.; McTT by Jang et al.

Quick refresher

Suppose we want to prove canonicity by induction.

Quick refresher

Suppose we want to prove canonicity by induction.

Easy case: computation on first-order types

$$\vdash \text{if } b \text{ then } n \text{ else } m \ : \ \mathsf{Nat}$$

Quick refresher

Suppose we want to prove canonicity by induction.

Easy case: computation on first-order types

$$\vdash \text{if } b \text{ then } n \text{ else } m \ : \ \text{Nat}$$

Look at the recursive result on b, return the correct recursive call among n and m.

Harder: higher-order types

$$\vdash f\,n : \mathsf{Nat}$$

Harder: higher-order types

$$\vdash f\,n : \mathrm{Nat}$$

f itself is responsible for the computation.

The recursive call on f should return

$\forall a, P_{\mathrm{Nat}}(n) \rightarrow P_{\mathrm{Nat}}(f\,n)$!

Canonicity on terms depend on the canonicity on types!

Canonicity on terms depend on the canonicity on types!

For $\vdash a : A$, we'd like

$$A_{\mathrm{rel}} : [\![A]\!]$$

$$a_{\mathrm{rel}} : [\![a]\!]_{A_{\mathrm{rel}}}$$

And more generally, for $\Gamma \vdash a : A$

$$\Gamma_{\mathrm{rel}} : [\![\Gamma]\!]$$

$$A_{\mathrm{rel}} : \forall \gamma, (\gamma_{\mathrm{rel}} : [\![\gamma]\!]_{\Gamma_{\mathrm{rel}}}) \to [\![A[\gamma]]\!]$$

$$a_{\mathrm{rel}} : \forall \gamma, (\gamma_{\mathrm{rel}} : [\![\gamma]\!]_{\Gamma_{\mathrm{rel}}}) \to [\![a[\gamma]]\!]_{A_{\mathrm{rel}}(\gamma_{\mathrm{rel}})}$$

That's the *fundamental lemma* of logical relations.

We also need a corresponding realizer for type and term conversions, $[\![A \equiv B]\!]$ and $[\![a \equiv b]\!]_{A_{\mathrm{rel}}}$.

We also need a corresponding realizer for type and term conversions, $[\![A \equiv B]\!]$ and $[\![a \equiv b]\!]_{A_{\mathrm{rel}}}$.

You can actually save some work and only define conversion realizers as a *partial equivalence relation* with $[\![A]\!] := [\![A \equiv A]\!]$.

What about the universe?

In bare MLTT, the universe is left underspecified.

We have a choice in the logical relation when defining $[\![\cdot]\!]$:

Positive	Negative

What about the universe?

In bare MLTT, the universe is left underspecified.

We have a choice in the logical relation when defining $[\![\cdot]\!]$:

Positive	Negative
Inductive of codes	Record of relations

What about the universe?

In bare MLTT, the universe is left underspecified.

We have a choice in the logical relation when defining $[\![\cdot]\!]$:

Positive	Negative
Inductive of codes	Record of relations
Limited to internal types	Can contain external types

What about the universe?

In bare MLTT, the universe is left underspecified.

We have a choice in the logical relation when defining $[\![\cdot]\!]$:

Positive	Negative
Inductive of codes	Record of relations
Limited to internal types	Can contain external types

We can feed the realizer of $\vdash f : \forall (A : \mathrm{U}), A \to A$ a specific relation to get a parametricity result.

What about the universe?

In bare MLTT, the universe is left underspecified.

We have a choice in the logical relation when defining $[\![\cdot]\!]$:

Positive	Negative
Inductive of codes	Record of relations
Limited to internal types	Can contain external types
Easy to formalize	Dependent PER hell?

Too easy... right?

In the end, we just defined a (terminating) evaluator in the meta-theory!

Too easy... right?

In the end, we just defined a (terminating) evaluator in the meta-theory!

But we have no guarantees about correctness! If $[\![\, b \,]\!]_{\mathsf{Bool}_{\mathsf{rel}}}$ tells me b is true, I want a witness of that!

We could prove correctness after the fact, but it's usually neater to make it correct-by-construction.

→ Let's just add some information in our logical relation!

isTrue : $[\![b]\!]_{\mathrm{Bool}_{\mathrm{rel}}}$

$$\mathsf{isTrue} : b \rightsquigarrow \mathsf{true} \to \quad [\![b]\!]_{\mathsf{Bool}_{\mathsf{rel}}}$$

Doesn't give us a derivation.

$$\mathsf{isTrue} : \cdot \vdash b \equiv \mathsf{true} \to [\![\, b \,]\!]_{\mathsf{Bool}_{\mathsf{rel}}}$$

Doesn't prove reduction works.

logrel-coq abstracts over the possible extra info we add to the logical relation.

logrel-coq abstracts over the possible extra info we add to the logical relation.

Axiomatizing precisely the bits we need for the fundamental theorem is difficult.

Instantiating the logical relation

A good candidate for our extra info: derivations for an algorithmic typing system.

$$\dfrac{\dfrac{\dots}{\vdash A \to B \equiv M} \qquad \dfrac{\dots}{\vdash M \equiv A' \to B'}}{\vdash A \to B \equiv A' \to B'}\ \text{trans}$$

Instantiating the logical relation

A good candidate for our extra info: derivations for an algorithmic typing system.

$$\dfrac{\dfrac{\cdots}{\vdash A \to B \equiv M} \qquad \dfrac{\cdots}{\vdash M \equiv A' \to B'}}{\vdash A \to B \equiv A' \to B'}\ \text{trans}$$

$$\Big\downarrow{\scriptstyle \text{LogRel}}$$

$$\dfrac{\dfrac{\cdots}{\vdash A \equiv A'} \qquad \dfrac{\cdots}{\vdash B \equiv B'}}{\vdash A \to B \equiv A' \to B'}\ \text{cong-}\Pi$$

Instantiating the logical relation

A good candidate for our extra info: derivations for an algorithmic typing system.

$$\dfrac{\dfrac{\dots}{\vdash A \to B \equiv M} \qquad \dfrac{\dots}{\vdash M \equiv A' \to B'}}{\vdash A \to B \equiv A' \to B'}\ \text{trans}$$

$$\Big\downarrow \text{LogRel}$$

Not so fast!

$$\dfrac{\dfrac{\dots}{\vdash A \equiv A'} \qquad \dfrac{\dots}{\vdash B \equiv B'}}{\vdash A \to B \equiv A' \to B'}\ \text{cong-}\Pi$$

The algorithmic instance of generic typing in logrel-coq is chimeric, containing declarative parts.

The algorithmic instance of generic typing in logrel-coq is chimeric, containing declarative parts.

We need to:

The algorithmic instance of generic typing in logrel-coq is chimeric, containing declarative parts.

We need to:

1. Instantiate with a dumbed down algorithmic system;

The algorithmic instance of generic typing in logrel-coq is chimeric, containing declarative parts.

We need to:

1. Instantiate with a dumbed down algorithmic system;
2. Deduce that the full algorithmic system satisfies the interface;

The algorithmic instance of generic typing in logrel-coq is chimeric, containing declarative parts.

We need to:

1. Instantiate with a dumbed down algorithmic system;
2. Deduce that the full algorithmic system satisfies the interface;
3. Instantiate with the full algorithmic system.

The algorithmic instance of generic typing in logrel-coq is chimeric, containing declarative parts.

We need to:

1. Instantiate with a dumbed down algorithmic system;
2. Deduce that the full algorithmic system satisfies the interface;
3. Instantiate with the full algorithmic system.

Feels unsatisfactory.

The abstract mentions an alternative free-standing system that types terms by first normalizing them.

The abstract mentions an alternative free-standing system that types terms by first normalizing them.

$$\frac{\vdash (\lambda(n : \mathsf{Bool}).\, n)\, \mathsf{zero} \rightsquigarrow \quad \mathsf{zero} : \mathsf{Nat} \qquad \vdash \mathsf{zero} : \mathsf{Nat}}{\vdash (\lambda(n : \mathsf{Bool}).\, n)\, \mathsf{zero} : \mathsf{Nat}}$$

The abstract mentions an alternative free-standing system that types terms by first normalizing them.

$$\frac{\vdash (\lambda(n : \mathsf{Bool}).\,n)\,\mathsf{zero} \rightsquigarrow \quad \mathsf{zero} : \mathsf{Nat} \qquad \vdash \mathsf{zero} : \mathsf{Nat}}{\vdash (\lambda(n : \color{red}{\mathsf{Bool}}).\,n)\,\color{red}{\mathsf{zero}} : \mathsf{Nat}}$$

The abstract mentions an alternative free-standing system that types terms by first normalizing them.

$$\frac{\vdash (\lambda(n : \mathsf{Bool}).\,n)\,\mathsf{zero} \rightsquigarrow_{\mathsf{wt}} \mathsf{zero} : \mathsf{Nat} \qquad \vdash \mathsf{zero} : \mathsf{Nat}}{\vdash (\lambda(n : \mathsf{Bool}).\,n)\,\mathsf{zero} : \mathsf{Nat}}$$

The abstract mentions an alternative free-standing system that types terms by first normalizing them.

$$\frac{\vdash (\lambda(n : \mathsf{Bool}).\,n)\,\mathsf{zero} \rightsquigarrow_{\mathsf{wt}} \mathsf{zero} : \mathsf{Nat} \qquad \vdash \mathsf{zero} : \mathsf{Nat}}{\vdash (\lambda(n : \mathsf{Bool}).\,n)\,\mathsf{zero} : \mathsf{Nat}}$$

→ Lots of redundant work.

Better served by a form of algorithmic typing followed by reflexivity for algorithmic conversion.

Conclusion

We're far from being able to match the literature on logical relations for theoretical and practical reasons.

A lot of refactoring is needed if we want to tackle more advanced systems efficiently.

Thanks for your attention!

Heterogeneous judgements

Completely heterogeneous judgements are nice to work with, since they avoid arbitrary choices!

Heterogeneous judgements

Completely heterogeneous judgements are nice to work with, since they avoid arbitrary choices!

Instead of

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, (x : ?) \vdash B}{\Gamma \vdash \Pi x : A.B \equiv \Pi x : A'.B}$$

Heterogeneous judgements

Completely heterogeneous judgements are nice to work with, since they avoid arbitrary choices!

Instead of

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, (x : {\color{red}?}) \vdash B}{\Gamma \vdash \Pi x : A.B \equiv \Pi x : A'.B}$$

We have

$$\frac{\Gamma \equiv \Delta \vdash A \equiv A' \qquad \Gamma, (x : A) \equiv \Delta, (x : A') \vdash B \equiv B'}{\Gamma \equiv \Delta \vdash \Pi x : A.B \equiv \Pi x : A'.B'}$$

Single mutual inductive

Reify different judgements as an inductive, and index derivations with it.

```
“Γ ⊢ t : A” : judgement
derivation  : judgement → Type
```

Single mutual inductive

Reify different judgements as an inductive, and index derivations with it.

```
“Γ ⊢ t : A” : judgement
derivation  : judgement → Type
```

Avoids `Combined Scheme` and meta-programming!