Extending SortPoly with Elimination Constraints in Rocq

The Rocqshop 2025, Reykjavik, Iceland

Tomás Díaz[1] Kenji Maillard[2] Johann Rosain[3] Matthieu Sozeau[2] Nicolas Tabareau[2] Éric Tanter[1] Théo Winterhalter[4]

September 27, 2025

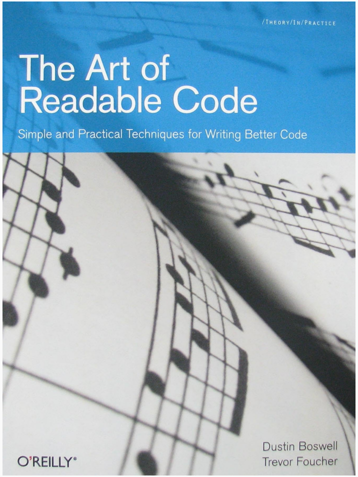
[1]PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile

[2]Galinette Project Team, LS2N & Inria de l'Université de Rennes, Nantes, France

[3]École Normale Supérieure de Lyon, Lyon, France

[4]LMF & Inria Saclay, Saclay, France

When You Stare Into The Code, it Stares Back At You



When You Stare Into The Code, it Stares Back At You

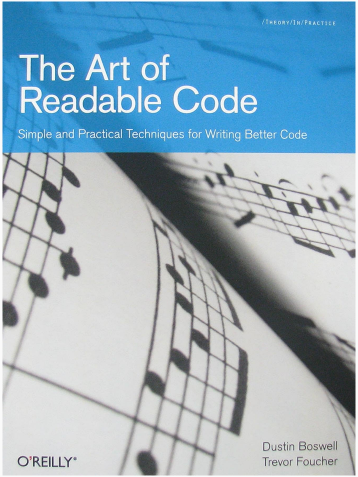


Figure 1. Book Rocq's developers forgot to read.

When You Stare Into The Code, it Stares Back At You

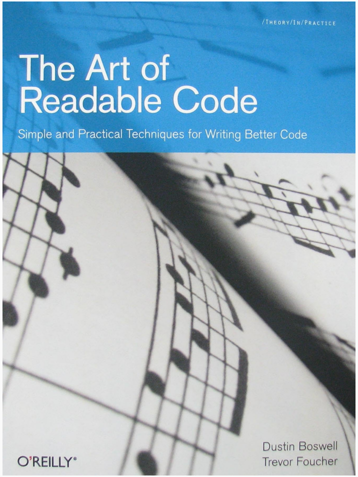


Figure 1. Book Rocq's developers forgot to read.

Unexpectedly:

Not a talk about Rocq's source code.

When You Stare Into The Code, it Stares Back At You

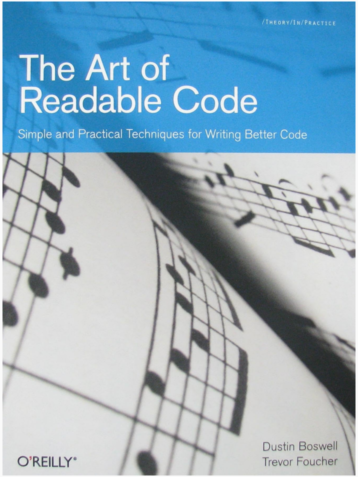


Figure 1. Book Rocq's developers forgot to read.

Unexpectedly:

Not a talk about Rocq's source code.

One of the big principles:

Don't repeat yourself.

Guess what?

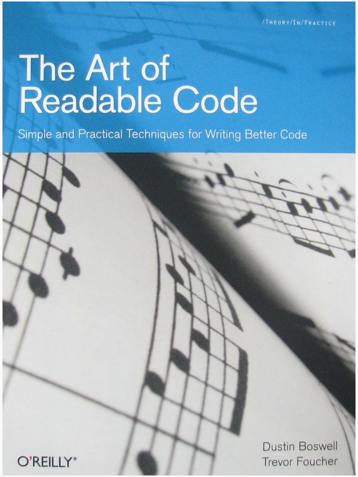


Figure 1. Book Rocq's developers forgot to read.

Unexpectedly:

Not a talk about Rocq's source code.

One of the big principles:

Don't repeat yourself.

Guess what? today's talk: duplications.

Déjà Vu

```
Inductive sum (A B : Type) : Type :=
| inl : A → sum A B
| inr : B → sum A B.
```

```
Inductive sum (A B : Type) : Type :=
| inl : A → sum A B
| inr : B → sum A B.

Inductive or (A B : Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.
```

Déjà Vu

```
Inductive sum (A B : Type) : Type :=
| inl : A → sum A B
| inr : B → sum A B.

Inductive or (A B : Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.

Inductive sumbool (A B : Prop) : Type :=
| left : A → sumbool A B
| right : B → sumbool A B.
```

Déjà Vu

```
Inductive sum (A B : Type) : Type :=
| inl : A → sum A B
| inr : B → sum A B.

Inductive or (A B : Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.

Inductive sumbool (A B : Prop) : Type :=
| left : A → sumbool A B
| right : B → sumbool A B.

Inductive sumor (A : Type) (B : Prop) :
    Type :=
| inleft : A → sumor A B
| inright : B → sumor A B.
```

Déjà Vu

```
Inductive sum (A B : Type) : Type :=
| inl : A → sum A B
| inr : B → sum A B.

Inductive or (A B : Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.

Inductive sumbool (A B : Prop) : Type :=
| left : A → sumbool A B
| right : B → sumbool A B.

Inductive sumor (A : Type) (B : Prop) :
    Type :=
| inleft : A → sumor A B
| inright : B → sumor A B.
```

Not All Heroes Wear Capes

Poiret et al. saved us from this world of suffering with SortPoly:

```
Inductive sum@{sl sr s ; ul ur}
  (A : 𝒰@{sl ; ul}) (B : 𝒰@{sr ; ur}) : 𝒰@{s ; max(ul, ur)} :=
| inl : A → sum A B
| inr : B → sum A B.
```

Not All Heroes Wear Capes

Poiret et al. saved us from this world of suffering with SortPoly:

```
Inductive sum@{sl sr s ; ul ur}
  (A : 𝒰@{sl ; ul}) (B : 𝒰@{sr ; ur}) : 𝒰@{s ; max(ul, ur)} :=
| inl : A → sum A B
| inr : B → sum A B.
```

"Universe Levels"

- Universe level polymorphism: Sozeau and Tabareau, 2014.

Not All Heroes Wear Capes

Poiret et al. saved us from this world of suffering with SortPoly:

```
Inductive sum@{sl sr s ; ul ur}
  (A : 𝒰@{sl ; ul}) (B : 𝒰@{sr ; ur}) : 𝒰@{s ; max(ul, ur)} :=
| inl : A → sum A B
| inr : B → sum A B.
```

“Sorts”

- Universe level polymorphism: Sozeau and Tabareau, 2014.
- In 2025, Poiret et al. bring sort polymorphism to Rocq.

Not All Heroes Wear Capes

Poiret et al. saved us from this world of suffering with SortPoly:

```
Inductive sum@{sl sr s ; ul ur}
  (A : 𝒰@{sl ; ul}) (B : 𝒰@{sr ; ur}) : 𝒰@{s ; max(ul, ur)} :=
| inl : A → sum A B
| inr : B → sum A B.
```

"Universes"

- Universe level polymorphism: Sozeau and Tabareau, 2014.
- In 2025, Poiret et al. bring sort polymorphism to Rocq.

Not All Heroes Wear Capes

Poiret et al. saved us from this world of suffering with SortPoly:

```
Inductive sum@{sl sr s ; ul ur}
  (A : 𝓤@{sl ; ul}) (B : 𝓤@{sr ; ur}) : 𝓤@{s ; max(ul, ur)} :=
| inl : A → sum A B
| inr : B → sum A B.
```

“Sorts” “Universe Levels”

“Universes”

- Universe level polymorphism: Sozeau and Tabareau, 2014.
- In 2025, Poiret et al. bring sort polymorphism to Rocq.

Not All Heroes Wear Capes

Poiret et al. saved us from this world of suffering with SortPoly:

```
Inductive sum@{sl sr s ; ul ur}
  (A : 𝒰@{sl ; ul}) (B : 𝒰@{sr ; ur}) : 𝒰@{s ; max(ul, ur)} :=
| inl : A → sum A B
| inr : B → sum A B.
```

"Sorts" "Universe Levels"

"Universes"

- Universe level polymorphism: Sozeau and Tabareau, 2014.
- In 2025, Poiret et al. bring sort polymorphism to Rocq.

But this is not enough to avoid duplication!

Arguably Worse Situation

With unbounded sort polymorphism: cannot define *e.g.*, a generic eliminator:

```
Definition sum_elim@{sl sr s s'}
  {A : 𝒰@{sl}} {B : 𝒰@{sr}} {C : 𝒰@{s'}} (u : sum@{sl sr s} A B)
  (f : A → C) (g : B → C) : C :=
  match u in sum@{sl sr s} A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

Arguably Worse Situation

With unbounded sort polymorphism: cannot define *e.g.*, a generic eliminator:

```
Definition sum_elim@{sl sr Prop Type}
  {A : 𝒰@{sl}} {B : 𝒰@{sr}} {C : 𝒰@{Type}} (u : sum@{sl sr Prop} A B)
  (f : A → C) (g : B → C) : C :=
  match u in sum@{sl sr Prop} A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

Otherwise, setting $\mathsf{s} := \texttt{Prop}$ and $\mathsf{s}' := \texttt{Type}$ makes Rocq inconsistent.

Arguably Worse Situation

With unbounded sort polymorphism: cannot define *e.g.*, a generic eliminator:

```
Definition sum_elim@{sl sr Prop Type}
  {A : 𝓤@{sl}} {B : 𝓤@{sr}} {C : 𝓤@{Type}} (u : sum@{sl sr Prop} A B)
  (f : A → C) (g : B → C) : C :=
  match u in sum@{sl sr Prop} A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

Otherwise, setting $s := $ `Prop` and $s' := $ `Type` makes Rocq inconsistent.

Still need to declare the different eliminators "by hand".

Shackling The Old Rooster

3 goals:

3 goals:

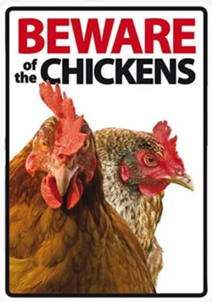
- use bounds to avoid duplication,

Shackling The Old Rooster

3 goals:

- use bounds to avoid duplication,
- preserve Rocq's consistency

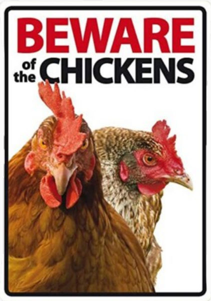
Shackling The Old Rooster



3 goals:

- use bounds to avoid duplication,
- preserve Rocq's consistency (otherwise the chicken is angry), and

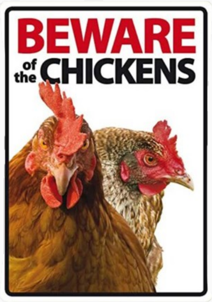
Shackling The Old Rooster



3 goals:

- use bounds to avoid duplication,
- preserve Rocq's consistency (otherwise the chicken is angry), and
- remove need of annotations.

Shackling The Old Rooster



3 goals:

- use bounds to avoid duplication,
- preserve Rocq's consistency (otherwise the chicken is angry), and
- remove need of annotations.

This is the job of elimination constraints in $\mathrm{SortPol}\vec{\mathrm{y}}$.

No Regression

For `s`, `s'` two *sorts*:

$$\texttt{s} \rightsquigarrow \texttt{s'} \iff \text{values in } \texttt{s'} \text{ can be produced by ones in } \texttt{s}$$

(via pattern-matching).

No Regression

For `s`, `s'` two *sorts*:

$$\texttt{s} \rightsquigarrow \texttt{s'} \iff \text{values in } \texttt{s'} \text{ can be produced by ones in } \texttt{s}$$

(via pattern-matching). Rocq's system:

$$\texttt{Type} \rightsquigarrow \texttt{Type}, \texttt{Type} \rightsquigarrow \texttt{Prop}, \texttt{Type} \rightsquigarrow \texttt{SProp},$$

No Regression

For `s`, `s'` two *sorts*:

$$\texttt{s} \rightsquigarrow \texttt{s'} \iff \text{values in } \texttt{s'} \text{ can be produced by ones in } \texttt{s}$$

(via pattern-matching). Rocq's system:

$$\texttt{Type} \rightsquigarrow \texttt{Type}, \texttt{Type} \rightsquigarrow \texttt{Prop}, \texttt{Type} \rightsquigarrow \texttt{SProp},$$
$$\texttt{Prop} \rightsquigarrow \texttt{Prop}, \texttt{Prop} \rightsquigarrow \texttt{SProp}, \text{and}$$

No Regression

For `s`, `s'` two *sorts*:

$$\texttt{s} \rightsquigarrow \texttt{s'} \iff \text{values in } \texttt{s'} \text{ can be produced by ones in } \texttt{s}$$

(via pattern-matching). Rocq's system:

$\texttt{Type} \rightsquigarrow \texttt{Type}$, $\texttt{Type} \rightsquigarrow \texttt{Prop}$, $\texttt{Type} \rightsquigarrow \texttt{SProp}$,
$\texttt{Prop} \rightsquigarrow \texttt{Prop}$, $\texttt{Prop} \rightsquigarrow \texttt{SProp}$, and
$\texttt{SProp} \rightsquigarrow \texttt{SProp}$.

No Regression

For `s`, `s'` two *sorts*:

$$\texttt{s} \rightsquigarrow \texttt{s'} \iff \text{values in } \texttt{s'} \text{ can be produced by ones in } \texttt{s}$$

(via pattern-matching). Rocq's system:

`Type` $\rightsquigarrow$ `Type`, `Type` $\rightsquigarrow$ `Prop`, `Type` $\rightsquigarrow$ `SProp`,
`Prop` $\rightsquigarrow$ `Prop`, `Prop` $\rightsquigarrow$ `SProp`, and
`SProp` $\rightsquigarrow$ `SProp`.

Study of the metatheory gives:

- consistency (under small condition), and
- transitivity.

```
Definition foo@{s | SProp ⇝ s} {A : 𝓤@{s}}
  (f : 𝔹@{SProp} → A) : f true = f false := eq_refl.
```

```
Definition foo@{s | SProp ⇝ s} {A : 𝒰@{s}}
  (f : 𝔹@{SProp} → A) : f true = f false := eq_refl.
```

Indeed, `true` $\equiv$ `false` in `SProp`, so `f true` $\equiv$ `f false`.

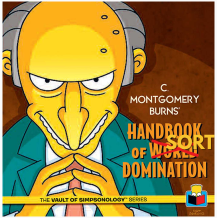
```
Definition foo@{s | SProp ⇝ s} {A : 𝓤@{s}}
  (f : 𝔹@{SProp} → A) : f true = f false := eq_refl.
```

Indeed, `true` $\equiv$ `false` in `SProp`, so `f true` $\equiv$ `f false`.

If unrestricted, undecidability is close (`Exc` exceptional sort, what happens with `Exc` $\rightsquigarrow$ `s` and `SProp` $\rightsquigarrow$ `s`)?

Everything You Never Wanted To Know About Domination

```
Definition foo@{s | SProp ⇝ s} {A : 𝓤@{s}}
  (f : 𝔹@{SProp} → A) : f true = f false := eq_refl.
```



Indeed, `true` $\equiv$ `false` in `SProp`, so `f true` $\equiv$ `f false`.

If unrestricted, undecidability is close (`Exc` exceptional sort, what happens with `Exc` $\rightsquigarrow$ `s` and `SProp` $\rightsquigarrow$ `s`)?

Dominant sort: unique minimal ground sort w.r.t. $\rightsquigarrow^{op}$.

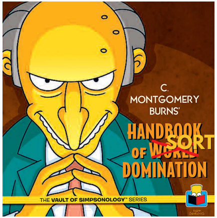
Everything You Never Wanted To Know About Domination

```
Definition foo@{s | SProp ⇝ s} {A : 𝒰@{s}}
  (f : 𝔹@{SProp} → A) : f true = f false := eq_refl.
```

Indeed, `true` $\equiv$ `false` in `SProp`, so `f true` $\equiv$ `f false`.

If unrestricted, undecidability is close (`Exc` exceptional sort, what happens with `Exc` $\rightsquigarrow$ `s` and `SProp` $\rightsquigarrow$ `s`)?

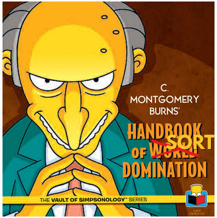
Dominant sort: unique minimal ground sort w.r.t. $\rightsquigarrow^{op}$.



initial sort + all sorts dominated $\Longrightarrow$ consistency.

Everything You Never Wanted To Know About Domination

```
Definition foo@{s | SProp ⇝ s} {A : 𝒰@{s}}
  (f : 𝔹@{SProp} → A) : f true = f false := eq_refl.
```



Indeed, `true` $\equiv$ `false` in `SProp`, so `f true` $\equiv$ `f false`.

If unrestricted, undecidability is close (`Exc` exceptional sort, what happens with `Exc` $\rightsquigarrow$ `s` and `SProp` $\rightsquigarrow$ `s`)?

Dominant sort: unique minimal ground sort w.r.t. $\rightsquigarrow^{op}$.

initial sort + all sorts dominated $\Longrightarrow$ consistency.

Good news: in Rocq, `Type` is initial.

$\Longrightarrow$ only need to ensure domination.

It’s Even Getting User-Friendly

“Transitive encoding” of elimination constraints that preserves typing.

It’s Even Getting User-Friendly

“Transitive encoding” of elimination constraints that preserves typing.

We hard-wire transitivity in the system. Benefits:

1. more concise declaration of elimination constraints,
2. catch more inconsistent stuff.

It's Even Getting User-Friendly

"Transitive encoding" of elimination constraints that preserves typing.

We hard-wire transitivity in the system. Benefits:

1. more concise declaration of elimination constraints,
2. catch more inconsistent stuff.

1: Rocq's system is refl. closure of `Type` $\rightsquigarrow$ `Prop`, `Prop` $\rightsquigarrow$ `SProp`.

2: Catch inadvertently-introduced `Prop` $\rightsquigarrow$ `Type` or non-dominated sort.

It's Even Getting User-Friendly

"Transitive encoding" of elimination constraints that preserves typing.

We hard-wire transitivity in the system. Benefits:

1. more concise declaration of elimination constraints,
2. catch more inconsistent stuff.

1: Rocq's system is refl. closure of `Type` $\rightsquigarrow$ `Prop`, `Prop` $\rightsquigarrow$ `SProp`.

2: Catch inadvertently-introduced `Prop` $\rightsquigarrow$ `Type` or non-dominated sort.

Does not catch everything, *e.g.*, inconsistent sort eliminating to `Type`.

Good But Not Great

Can write the generic eliminator:

```
Definition sum_elim@{sl sr s s' | s ⇝ s'}
  {A : 𝓤@{sl}} {B : 𝓤@{sr}} {C : 𝓤@{s'}} (u : sum@{sl sr s} A B)
  (f : A → C) (g : B → C) : C :=
  match u in sum@{sl sr s} A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

Good But Not Great

Can write the generic eliminator:

```
Definition sum_elim@{sl sr s s' | s ⇝ s'}
  {A : 𝓤@{sl}} {B : 𝓤@{sr}} {C : 𝓤@{s'}} (u : sum@{sl sr s} A B)
  (f : A → C) (g : B → C) : C :=
  match u in sum@{sl sr s} A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

Small problem: this many annotations drive one crazy!

My face after annotating terms for 15 minutes

Good But Not Great

Can write the generic eliminator:

```
Definition sum_elim@{sl sr s s' | s ⇝ s'}
  {A : 𝓤@{sl}} {B : 𝓤@{sr}} {C : 𝓤@{s'}} (u : sum@{sl sr s} A B)
  (f : A → C) (g : B → C) : C :=
  match u in sum@{sl sr s} A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

My face after annotating terms for 15 minutes

Small problem: this many annotations drive one crazy!

Solution: one elaboration procedure to infer them all.

But Here, There *is* a Catharsis

2 good news:

- $\mathrm{SortPol\hat{\vec{y}}}$ enjoys principality.
- Have an elaboration procedure yielding the most generic term.

But Here, There *is* a Catharsis

2 good news:

- $\mathrm{SortPoly}^{\overrightarrow{\forall}}$ enjoys principality.
- Have an elaboration procedure yielding the most generic term.

Concretely, for the user:

```
Set Universe Polymorphism.
Definition sum_elim {A B C : Type}
  (u : sum A B) (f : A → C) (g : B → C) : C :=
  match u in sum A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

yields the principal term of the previous slide.

But Here, There *is* a Catharsis

2 good news:

- $\mathrm{SortPoly}^{\vec{\curlyvee}}$ enjoys principality.
- Have an elaboration procedure yielding the most generic term.

Concretely, for the user:

```
Set Universe Polymorphism.
Definition sum_elim {A B C : Type}
  (u : sum A B) (f : A → C) (g : B → C) : C :=
  match u in sum A B return C with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

Phew, we won't have to go insane over that

yields the principal term of the previous slide.

There is Even Actual Work Done

Implementation of elimination constraints in Rocq:

- reuse of the universe level graph for transitive closure,
- ad-hoc checks for dominant sorts (amortized constant complexity),
- ad-hoc checks to avoid introducing unwanted constraints,
- manual prohibition of `SProp` $\rightsquigarrow$ `s`.

There is Even Actual Work Done

Implementation of elimination constraints in Rocq:

- reuse of the universe level graph for transitive closure,
- ad-hoc checks for dominant sorts (amortized constant complexity),
- ad-hoc checks to avoid introducing unwanted constraints,
- manual prohibition of `SProp` $\rightsquigarrow$ `s`.

Expect some performance regressions in the monomorphic case:

- eliminability check through a graph,
- bigger structures at elaboration.

Rough Planning (There is a Strange Theme in These Titles, no?)

Current phase:

- Develop the constraint graph, and plug it at the right places.
- Parsing with annotations.

Check out the RFC!

Rough Planning (There is a Strange Theme in These Titles, no?)

Current phase:

- Develop the constraint graph, and plug it at the right places.
- Parsing with annotations.

Next phase:

- Automatic elaboration if flag `Set Universe Polymorphism`.
- Relax restriction on primitive records.
- Automatic generation of generic induction scheme.

Check out the RFC!

Rough Planning (There is a Strange Theme in These Titles, no?)

Current phase:

- Develop the constraint graph, and plug it at the right places.
- Parsing with annotations.

Next phase:

- Automatic elaboration if flag `Set Universe Polymorphism`.
- Relax restriction on primitive records.
- Automatic generation of generic induction scheme.

Near future:

- Make use of dominant sorts in conversion tests.

Check out the RFC!

This is Goodbye (For Now)

Want to play with elimination constraints? Latest development version:

This is Goodbye (For Now)

Want to play with elimination constraints? Latest development version:

Real conclusion:

Embrace SortPoly, it is painless for users.*

*Except for mad people annotating stuff.

This is Goodbye (For Now)

Want to play with elimination constraints? Latest development version:

Real conclusion:

Embrace SortPoly, it is painless for users.*

*Except for mad people annotating stuff.

Any question(s)?