Extending Sort Polymorphism with Elimination Constraints

TYPES2025, Glasgow

Tomás Díaz[1] Johann Rosain[2] Matthieu Sozeau[2]
Nicolas Tabareau [2] Théo Winterhalter [3]

June 10, 2025

[1]University of Chile, Chile

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

[3]LMF & Inria Saclay, Saclay, France

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
 match eq_decℕ (val x) (val y) with
 | inl e ⇒ inl (f_equal Var e)
 | inr f ⇒ inr (fun e ⇒ f (f_equal val e))
 end.
```

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
 match eq_decℕ (val x) (val y) with
 | inl e ⇒ inl (f_equal Var e)
 | inr f ⇒ inr (fun e ⇒ f (f_equal val e))
 end.
```

A Short Metastory

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
 match eq_decℕ (val x) (val y) with
 | inl e ⇒ inl (f_equal Var e)
 | inr f ⇒ inr (fun e ⇒ f (f_equal val e))
 end.
```

A Short Metastory

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
  match eq_decℕ (val x) (val y) with
  | inl e ⇒ inl (f_equal Var e)
  | inr f ⇒ inr (fun e ⇒ f (f_equal val e))
  end.
```

Error: Found a constructor of inductive type `sum` while a constructor of `sumbool` is expected.

A Short Metastory

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
 match eq_decℕ (val x) (val y) with
 | inleft e ⇒ inleft (f_equal Var e)
 | inright f ⇒ inright (fun e ⇒ f (f_equal val e))
 end.
```

A Short Metastory

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
  match eq_decℕ (val x) (val y) with
  | inleft e ⇒ inleft (f_equal Var e)
  | inright f ⇒ inright (fun e ⇒ f (f_equal val e))
  end.
```

Error: Found a constructor of inductive type `sumor` while a constructor of `sumbool` is expected.

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Print sumbool.

Inductive sumbool (A B : Prop) : Type@{0} :=
| left : A → {A} + {B}
| right : B → {A} + {B}.
```

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
 match eq_decℕ (val x) (val y) with
 | left e ⇒ left (f_equal Var e)
 | right f ⇒ right (fun e ⇒ f (f_equal val e))
 end.
```

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
  match eq_decℕ (val x) (val y) with
  | left e ⇒ left (f_equal Var e)
  | right f ⇒ right (fun e ⇒ f (f_equal val e))
  end.
```

A Short Metastory

Real situation™

```
#[projections(primitive)] Record QVar : Type@{0} := mkQVar { val : ℕ }.
Definition Var (n : ℕ) : QVar := {| val := n |}.

Definition eq_dec_qvar (x y : QVar) : { x = y } + { x ≠ y } :=
  match eq_decℕ (val x) (val y) with
  | left e ⇒ left (f_equal Var e)
  | right f ⇒ right (fun e ⇒ f (f_equal val e))
  end.
```

Pfiou.

Why Would You Do That?

Rocq's sorts: `SProp`, `Prop`, `Type`.

Powerful, but at what cost?

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

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

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

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

(This slide is powered by Rocq's `Corelib`©.)

Humanity's Savior: ~~Jesus Christ~~ Sort Polymorphism

Since Coq 8.19:

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

Recall: a sort is a quotiented pair (quality, universe)

```
Definition sum@{u v} := psum@{Type Type Type ; u v}.
 (* sum : Type@{u} → Type@{v} → Type@{max(u,v)} *)
Definition or := psum@{Prop Prop Prop ; 0 0}.
 (* or : Prop → Prop → Prop *)
Definition sumbool := psum@{Prop Prop Type ; 0 0}.
 (* sumbool : Prop → Prop → Type@{0} *)
Definition sumor@{u} := psum@{Type Prop Type ; u 0}.
 (* sumor : Type@{u} → Prop → Type@{u} *)
```

Humanity's Savior: ~~Jesus Christ~~ Sort Polymorphism

Since Coq 8.19:

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

Recall: a sort is a quotiented pair (quality, universe)

```
Definition sum@{u v} := psum@{Type Type Type ; u v}.
 (* sum : Type@{u} → Type@{v} → Type@{max(u,v)} *)
Definition or := psum@{Prop Prop Prop ; 0 0}.
 (* or : Prop → Prop → Prop *)
Definition sumbool := psum@{Prop Prop Type ; 0 0}.
 (* sumbool : Prop → Prop → Type@{0} *)
Definition sumor@{u} := psum@{Type Prop Type ; u 0}.
 (* sumor : Type@{u} → Prop → Type@{u} *)
```

```
psum is defined
```

Wait, what?

Mild Disappointment

```
Definition psum_elim@{sl sr s s′ ; ul ur}
 {A : 𝒰@{sl ; ul}} {B : 𝒰@{sr ; ur}} {C : 𝒰@{s′ ; max(ul,ur)}}
 (f : A → C) (g : B → C) (x : psum@{sl sr s ; ul ur} A B) :=
 match x with
 | inl a ⇒ f a
 | inr b ⇒ g b
 end.
```

Mild Disappointment

```
Definition psum_elim@{sl sr s s' ; ul ur}
 {A : 𝒰@{sl ; ul}} {B : 𝒰@{sr ; ur}} {C : 𝒰@{s' ; max(ul,ur)}}
 (f : A → C) (g : B → C) (x : psum@{sl sr s ; ul ur} A B) :=
 match x with
 | inl a ⇒ f a
 | inr b ⇒ g b
 end.
```

Error: Elimination of a sort polymorphic inductive object instantiated to a variable sort quality is only allowed on a predicate in the same sort quality.

Mild Disappointment

```
Definition psum_elim@{sl sr s s' ; ul ur}
  {A : 𝒰@{sl ; ul}} {B : 𝒰@{sr ; ur}} {C : 𝒰@{s' ; max(ul,ur)}}
  (f : A → C) (g : B → C) (x : psum@{sl sr s ; ul ur} A B) :=
  match x with
  | inl a ⇒ f a
  | inr b ⇒ g b
  end.
```

Error: Elimination of a sort polymorphic inductive object instantiated to a variable sort quality is only allowed on a predicate in the same sort quality.

$\implies$ Have to declare eliminators by hand...[1]

[1] With only `Prop` and `Type`, we could do something using bad dark magic though.

Today's Talk Objective

```
Definition psum_elim@{sl sr s s' ; ul ur} {A : 𝒰@{sl ; ul}}
 {B : 𝒰@{sr ; ur}} {C : 𝒰@{s' ; max(ul,ur)}} (f : A → C) (g : B → C)
 (x : psum@{sl sr s ; ul ur} A B) :=
 match x with
 | inl a ⇒ f a
 | inr b ⇒ g b
 end.

Definition sumbool_rect := psum_elim@{Prop Prop Type Type;0 0}.
Definition sumbool_ind := psum_elim@{Prop Prop Type Prop;0 0}.
Definition sumbool_sind := psum_elim@{Prop Prop Type SProp;0 0}.

Fail Definition or_rect := psum_elim@{Prop Prop Prop Type;0 0}.
```

The command has indeed failed with message: cannot create computational content from proof.

Overview of Elimination in Rocq's Type Theory

Binary relation "eliminates to":

- `Type` eliminates to `Prop`,
- `Type` eliminates to `SProp`,
- `Prop` eliminates to `SProp`,
- $\mathtt{s}$ eliminates to $\mathtt{s}$.

$$\frac{\dots \qquad \mathtt{x : A} \qquad \mathtt{A : s} \qquad \mathtt{P : s'} \qquad \mathtt{s\ eliminates\ to\ s'}}{\mathtt{match}\ \mathtt{x}\ \mathtt{return}\ \mathtt{P}\ \mathtt{with}\ \dots\ \mathtt{end}\ \text{well typed}}$$

$$\frac{\dots \qquad \mathtt{x : A} \qquad \mathtt{A : s} \qquad \mathtt{P : s'} \qquad \mathtt{s\ eliminates\ to\ s'\ or\ s = Prop}}{\mathtt{fix}\ \mathtt{F}\ \mathtt{x}\ \vec{y}\ \{\mathtt{struct}\ \mathtt{x}\} : \mathtt{P} := \dots\ \text{well typed}}$$

Rocq's (Updated) Type Theory

Populate partial order $\rightsquigarrow$:

$$\text{User-defined } \mathtt{s} \rightsquigarrow \mathtt{s'}, \qquad \mathtt{s}\ \mathtt{eliminates}\ \mathtt{to}\ \mathtt{s'} \implies \mathtt{s} \rightsquigarrow \mathtt{s'}.$$

Populate partial order $\rightsquigarrow$:

$$\text{User-defined } \mathtt{s} \rightsquigarrow \mathtt{s}', \qquad \mathtt{s}\ \mathtt{eliminates}\ \mathtt{to}\ \mathtt{s}' \implies \mathtt{s} \rightsquigarrow \mathtt{s}'.$$

$$\frac{\ldots \qquad \mathtt{x} : \mathtt{A} \qquad \mathtt{A} : \mathtt{s} \qquad \mathtt{P} : \mathtt{s}' \qquad \mathtt{s} \rightsquigarrow \mathtt{s}'}{\mathtt{match}\ \mathtt{x}\ \mathtt{return}\ \mathtt{P}\ \mathtt{with} \ldots \mathtt{end}\ \text{well typed}}$$

$$\frac{\ldots \qquad \mathtt{x} : \mathtt{A} \qquad \mathtt{A} : \mathtt{s} \qquad \mathtt{P} : \mathtt{s}' \qquad \mathtt{s} \rightsquigarrow \mathtt{s}'\ \mathtt{or}\ \mathtt{s} \rightsquigarrow \mathtt{Prop}}{\mathtt{fix}\ \mathtt{F}\ \mathtt{x}\ \vec{y}\ \{\mathtt{struct}\ \mathtt{x}\} : \mathtt{P} := \ldots\ \text{well typed}}$$

For Once, It's Not My Fault

```
Definition foo@{s|s ⇝ Prop} ... := ...
Definition bar@{s′ s″|s′ ⇝ s″} ... := ...

Sort Exc. (* Global sort *)
Axiom raise : ∀ A : Exc, A.
Constraint Type ⇝ Exc.

Definition foo'@{s s′|Exc ⇝ s, s ⇝ s′, s′ ⇝ Prop} :=
 ... foo@{s′} ... bar@{s s′} ...
```

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

For Once, It’s Not My Fault

```
Definition foo@{s|s ⇝ Prop} ... := ...
Definition bar@{s′ s″|s′ ⇝ s″} ... := ...

Sort Exc. (* Global sort *)
Axiom raise : ∀ A : Exc, A.
Constraint Type ⇝ Exc.

Definition foo’@{s s′|Exc ⇝ s, s ⇝ s′, s′ ⇝ Prop} :=
 ... foo@{s′} ... bar@{s s′} ...
```

$$\begin{tikzcd}
\mathtt{Type} \arrow[d, rightsquigarrow] & \\
\mathtt{Prop} \arrow[d, rightsquigarrow] & \\
\mathtt{SProp} & \mathtt{s} \arrow[ul, rightsquigarrow]
\end{tikzcd}$$

For Once, It’s Not My Fault

```
Definition foo@{s|s ⇝ Prop} ... := ...
Definition bar@{s′ s″|s′ ⇝ s″} ... := ...

Sort Exc. (* Global sort *)
Axiom raise : ∀ A : Exc, A.
Constraint Type ⇝ Exc.

Definition foo’@{s s′|Exc ⇝ s, s ⇝ s′, s′ ⇝ Prop} :=
  ... foo@{s′} ... bar@{s s′} ...
```

$$
\begin{tikzcd}
\mathtt{Type} \arrow[d, rightsquigarrow] & & \\
\mathtt{Prop} \arrow[d, rightsquigarrow] & & s' \arrow[d, rightsquigarrow] \\
\mathtt{SProp} & s \arrow[ul, rightsquigarrow] & s''
\end{tikzcd}
$$

For Once, It's Not My Fault

```
Definition foo@{s|s ⇝ Prop} ... := ...
Definition bar@{s′ s″|s′ ⇝ s″} ... := ...

Sort Exc. (* Global sort *)
Axiom raise : ∀ A : Exc, A.
Constraint Type ⇝ Exc.

Definition foo′@{s s′|Exc ⇝ s, s ⇝ s′, s′ ⇝ Prop} :=
  ... foo@{s′} ... bar@{s s′} ...
```

$$
\begin{tikzcd}
\mathtt{Type} \arrow[d, rightsquigarrow] & & \\
\mathtt{Prop} \arrow[d, rightsquigarrow] & \mathtt{Exc} & s' \arrow[d, rightsquigarrow] \\
\mathtt{SProp} & s \arrow[ul, rightsquigarrow] & s''
\end{tikzcd}
$$

For Once, It's Not My Fault

```
Definition foo@{s|s ⇝ Prop} ... := ...
Definition bar@{s′ s″|s′ ⇝ s″} ... := ...

Sort Exc. (* Global sort *)
Axiom raise : ∀ A : Exc, A.
Constraint Type ⇝ Exc.

Definition foo’@{s s′|Exc ⇝ s, s ⇝ s′, s′ ⇝ Prop} :=
  ... foo@{s′} ... bar@{s s′} ...
```

$$
\begin{tikzcd}
\mathsf{Type} \arrow[d, squiggly] \arrow[dr, squiggly] & & \\
\mathsf{Prop} \arrow[d, squiggly] & \mathsf{Exc} & s' \arrow[d, squiggly] \\
\mathsf{SProp} & s \arrow[ul, squiggly] & s''
\end{tikzcd}
$$

```
Definition foo@{s|s ⇝ Prop} ... := ...
Definition bar@{s′ s″|s′ ⇝ s″} ... := ...

Sort Exc. (* Global sort *)
Axiom raise : ∀ A : Exc, A.
Constraint Type ⇝ Exc.

Definition foo’@{s s′|Exc ⇝ s, s ⇝ s′, s′ ⇝ Prop} :=
  ... foo@{s′} ... bar@{s s′} ...
```

$$
\begin{tikzcd}
\mathtt{Type} \arrow[d, rightsquigarrow] \arrow[dr, rightsquigarrow] & & \\
\mathtt{Prop} \arrow[d, rightsquigarrow] & \mathtt{Exc} \arrow[r, rightsquigarrow] & \mathtt{s} \arrow[dl, rightsquigarrow] \\
\mathtt{SProp} & \mathtt{s}' \arrow[ul, rightsquigarrow] &
\end{tikzcd}
$$

Error: Elimination constraints imply an undeclared elimination between `Exc` and `Prop`.

For Once, It's Not My Fault

```
Definition foo@{s|s ⇝ Prop} ... := ...
Definition bar@{s′ s″|s′ ⇝ s″} ... := ...

Sort Exc. (* Global sort *)
Axiom raise : ∀ A : Exc, A.
Constraint Type ⇝ Exc.

Definition foo′@{s s′|Exc ⇝ s, s ⇝ s′, s′ ⇝ Prop} :=
  ... foo@{s′} ... bar@{s s′} ...
```

$$
\begin{tikzcd}
\mathtt{Type} \arrow[d, rightsquigarrow] \arrow[dr, rightsquigarrow] & & \\
\mathtt{Prop} \arrow[d, rightsquigarrow] & \mathtt{Exc} \arrow[r, rightsquigarrow] & s \arrow[dl, rightsquigarrow] \\
\mathtt{SProp} & s' \arrow[ul, rightsquigarrow] &
\end{tikzcd}
$$

Error: Elimination constraints imply an undeclared elimination between `Exc` and `Prop`.

Elimination in (S)`Prop`: consistency...

(Im)Predicativity

Definition

A sort `s` is impredicative if there is a term of type $\mathcal{U}$@{s;u} $\simeq$ $\mathcal{U}$@{s;v}.

`SProp` & `Prop` are impredicative.

Elimination	Provable properties
`Prop` $\rightsquigarrow$ `s`	`s` impredicative
`SProp` $\rightsquigarrow$ `s`	`s` impredicative & prop. proof irrelevant

But `s` predicative: could be `Type`...

(Im)Predicativity

Definition

A sort s is impredicative if there is a term of type $\mathcal{U}$@{s;u} $\simeq$ $\mathcal{U}$@{s;v}.

SProp & Prop are impredicative.

Elimination	Provable properties
Prop $\rightsquigarrow$ s	s impredicative
SProp $\rightsquigarrow$ s	s impredicative & prop. proof irrelevant

But s predicative: could be Type...

$\cancel{\text{Prop} \rightsquigarrow \text{s}}$ $\cancel{\text{SProp} \rightsquigarrow \text{s}}$

What If It Was Allowed?

Monomorphization theorem: for every

`Definition` $\texttt{foo@\{s s'|s} \rightsquigarrow \texttt{s'\}} := \ldots$

duplicate for *ground* $\texttt{s}$, $\texttt{s'}$ s.t. $\texttt{s} \rightsquigarrow \texttt{s'}$.

- Everything is well typed.
- Implies *ground* equiconsistency.

What If It Was Allowed?

Monomorphization theorem: for every

`Definition foo@{s s'|s` $\rightsquigarrow$ `s'} := ...`

duplicate for *ground* s, s′ s.t. s $\rightsquigarrow$ s′.

- Everything is well typed.
- Implies *ground* equiconsistency.

With global sorts:

- `SProp` $\rightsquigarrow$ s: acceptable.
- `Prop` $\rightsquigarrow$ s: under investigation.
- User-defined inconsistencies (e.g., `Exc` and `Exc` $\rightsquigarrow$ `Prop`).

Status of the Current Implementation

Can write $\mathsf{s} \rightsquigarrow \mathsf{s}'$ for *local* and *global* sorts. But:

	Current status	Solution?
Rigid sorts	User specifies all constraints	Doing otherwise is unsound Consistency is user's responsibility
`Prop` $\rightsquigarrow$ `s`	Prohibited	Predicativity mark (sufficient?)
`SProp` $\rightsquigarrow$ `s`	Prohibited	Definitional PI + predicativity mark

Status of the Current Implementation

Can write $\mathsf{s} \rightsquigarrow \mathsf{s}'$ for *local* and *global* sorts. But:

	Current status	Solution?
Rigid sorts	User specifies all constraints	Doing otherwise is unsound Consistency is user's responsibility
`Prop` $\rightsquigarrow$ s	Prohibited	Predicativity mark (sufficient?)
`SProp` $\rightsquigarrow$ s	Prohibited	Definitional PI + predicativity mark

Remark

(Sub)singleton elimination is not handled in the framework.

The Case of Cases

(Sub)singleton elimination: inductives with $\leqslant 1$ constructor: $\bot$, `Acc`essibility, `eq`uality, conjunction, ...

Long story short: I've bluffed you.

$$\frac{\ldots \qquad \mathtt{x} : \mathtt{A} \qquad \mathtt{A} : \mathtt{s} \qquad \mathtt{P} : \mathtt{s}' \qquad \mathtt{s} \rightsquigarrow \mathtt{s}'}{\mathtt{match}\ \mathtt{x}\ \mathtt{return}\ \mathtt{P}\ \mathtt{with} \ldots \mathtt{end}\ \text{well typed}}$$

The Case of Cases

(Sub)singleton elimination: inductives with $\leqslant 1$ constructor: $\bot$, `Acc`essibility, equality, conjunction, ...

Long story short: I've bluffed you.

$$\frac{\ldots \qquad \mathtt{x} : \mathtt{A} \qquad \mathtt{A} : \mathtt{s} \qquad \mathtt{P} : \mathtt{s}' \qquad \mathtt{s} \rightsquigarrow \mathtt{s}' \text{ or } \mathtt{s} \text{ has subsingleton elimination}}{\mathtt{match}\ \mathtt{x}\ \mathtt{return}\ \mathtt{P}\ \mathtt{with} \ldots \mathtt{end} \text{ well typed}}$$

(Sub)singleton elimination: inductives with $\leqslant 1$ constructor: $\perp$, `Accessibility`, equality, conjunction, ...

Long story short: I've bluffed you.

$$\frac{\begin{array}{c}\ldots \qquad \mathtt{x} : \mathtt{A} \qquad \mathtt{A} : \mathtt{s} \qquad \mathtt{P} : \mathtt{s}' \\ \mathtt{s} \rightsquigarrow \mathtt{s}' \,\mathtt{or\ s\ has\ subsingleton\ elimination}\end{array}}{\mathtt{match}\ \mathtt{x}\ \mathtt{return}\ \mathtt{P}\ \mathtt{with}\ \ldots\ \mathtt{end}\ \text{well typed}}$$

$$\frac{\begin{array}{c}\ldots \qquad \mathtt{x} : \mathtt{A} \\ \mathtt{A} : \mathtt{Record}\ \mathtt{in}\ \mathtt{s} \qquad \mathtt{P} : \mathtt{s}'\end{array}}{\mathtt{match'}\ \mathtt{x}\ \mathtt{return}\ \mathtt{P}\ \mathtt{with}\ \ldots\ \mathtt{end}\ \text{wt}} \qquad \frac{\begin{array}{c}\ldots \qquad \mathtt{x} : \mathtt{A} \qquad \mathtt{A} : \mathtt{Inductive}\ \mathtt{in}\ \mathtt{s} \\ \mathtt{P} : \mathtt{s}' \qquad \mathtt{s} \rightsquigarrow \mathtt{s}'\end{array}}{\mathtt{match}\ \mathtt{x}\ \mathtt{return}\ \mathtt{P}\ \mathtt{with}\ \ldots\ \mathtt{end}\ \text{wt}}$$

Conclusion

We have:

- Implemented elimination constraints in Rocq.
- Enabled writing the most generic eliminator in Rocq.
- Proved consistency of system (on paper, for *ground* sorts).

```
Definition psum_elim@{sl sr s s' ; ul ur | s ⇝ s'} {A : 𝒰@{sl ; ul}}
 {B : 𝒰@{sr ; ur}} {C : 𝒰@{s' ; max(ul,ur)}} (f : A → C) (g : B → C)
 (x : psum@{sl sr s ; ul ur} A B) :=
 match x with
 | inl a ⇒ f a
 | inr b ⇒ g b
 end.
```

Conclusion

We have:

- Implemented elimination constraints in Rocq.
- Enabled writing the most generic eliminator in Rocq.
- Proved consistency of system (on paper, for *ground* sorts).

```
Definition psum_elim@{sl sr s s′ ; ul ur | s ⇝ s′} {A : 𝒰@{sl ; ul}}
 {B : 𝒰@{sr ; ur}} {C : 𝒰@{s′ ; max(ul,ur)}} (f : A → C) (g : B → C)
 (x : psum@{sl sr s ; ul ur} A B) :=
 match x with
 | inl a ⇒ f a
 | inr b ⇒ g b
 end.
```

😎

Conclusion

We have:

- Implemented elimination constraints in Rocq.
- Enabled writing the most generic eliminator in Rocq.
- Proved consistency of system (on paper, for *ground* sorts).

We plan to:

- Add elimination constraints in MetaRocq (current status: formalizing sort polymorphism).
- Allow `SProp` $\rightsquigarrow$ `s`.
- Allow `Prop` $\rightsquigarrow$ `s` if consistent.

Conclusion

We have:

- Implemented elimination constraints in Rocq.
- Enabled writing the most generic eliminator in Rocq.
- Proved consistency of system (on paper, for *ground* sorts).

We plan to:

- Add elimination constraints in MetaRocq (current status: formalizing sort polymorphism).
- Allow `SProp` $\rightsquigarrow$ `s`.
- Allow `Prop` $\rightsquigarrow$ `s` if consistent.

Thank you for your attention!