Jason Gross' Wishlist for Coq

POPL 2014 — Coq Users Meeting

Judgmental Equality

More Powerful Judgmental Equality

Judgmental Equality

More Powerful Judgmental Equality

Warning: Some of my proposals get rather insane, so the further on in this section they are, the more grains of salt you should be taking them with.

Judgmental Equality

My Wishes: η for records

η for records

Implemented by Matthieu Sozeau; in 8.5, I can now have $(\mathcal{C}^{\text{op}})^{\text{op}} \equiv \mathcal{C}$ for categories $\mathcal{C}$!

Judgmental Equality

My Wishes: η for records

η for records

Implemented by Matthieu Sozeau; in 8.5, I can now have $(\mathcal{C}^{\mathrm{op}})^{\mathrm{op}} \equiv \mathcal{C}$ for categories $\mathcal{C}$!

It would still be nice to have

$$\forall \; \texttt{x} \; \texttt{y} \; \texttt{:} \; \texttt{unit,} \; \texttt{x} \equiv \texttt{tt} \equiv \texttt{y.}$$

Judgmental Equality

My Wishes: η for inductives

η for inductive types

I want

```
∀ A B (x : A + B),
  match x with
    | inl x' ⇒ inl x'
    | inr x' ⇒ inr x'
  end ≡ x
```

Judgmental Equality

My Wishes: η for inductives

η for inductive types

I want

```
∀ A (x y : A) (p : x = y),
  match p in (_ = y') return (x = y') with
    | eq_refl ⇒ eq_refl
  end ≡ p
```

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

I want a `match` to eat up unused arguments:

```
match p as p' in (T x _)
  return (T' x p' → T'' x p')
with
  | con1 ⇒ (λ _ ⇒ val1)
  ...
end y
≡
```

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

I want a `match` to eat up unused arguments:

$\equiv$

```
match p as p' in (T x _)
  return (T'' x p')
with
  | con1 ⇒ val1
  ...
end
```

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

And many more. . . (see Appendix)

Judgmental Equality

My Wishes: Judgmental Groupoid Laws

Judgmental Groupoid Laws

I want (the option of) `Types` to be strict ∞-groupoids

$$(p^{-1})^{-1} \equiv p \qquad (p^{-1} \text{ is } \texttt{eq\_sym } p)$$

$$p \circ (q \circ r) \equiv (p \circ q) \circ r \qquad (p \circ q \text{ is } \texttt{eq\_trans } p\ q)$$

$$p \circ 1 \equiv p \equiv 1 \circ p \qquad (1 \text{ is } \texttt{eq\_refl})$$

Judgmental Equality

My Wishes: Axiom K-based Pattern Matching When It's Provable

K-Based Pattern Matching

I want K-based pattern matching on types which Coq can infer are hSets (satisfy uniqueness of identity proofs, and therefore K), any maybe for types where I can prove K.

Judgmental Equality

My Wishes: Axiom K-based Pattern Matching When It's Provable

K-Based Pattern Matching

I want K-based pattern matching on types which Coq can infer are hSets (satisfy uniqueness of identity proofs, and therefore K), any maybe for types where I can prove K. Alternatively, maybe a "strict 0-truncation" operator, and support for K there.

Judgmental Equality

My Wishes: Axiom K-based Pattern Matching When It's Provable

K-Based Pattern Matching

I want K-based pattern matching on types which Coq can infer are hSets (satisfy uniqueness of identity proofs, and therefore K), any maybe for types where I can prove K. Alternatively, maybe a "strict 0-truncation" operator, and support for K there.

Proposal by Pierre Corbineau: "The K axiom in Coq (almost) for free"[1]

[1] http://coq.inria.fr/files/adt-2fev10-corbineau.pdf

Judgmental Equality

My Wishes: Irrelevant Types

Irrelevant Types

I want types with judgmental (proof) irrelevance, like dotted fields in Agda.

Judgmental Equality

My Wishes: Irrelevant Types

Irrelevant Types

I want types with judgmental (proof) irrelevance, like dotted fields in Agda. These are strict hProps.

Judgmental Equality

My Wishes: Irrelevant Types

Irrelevant Types

I want types with judgmental (proof) irrelevance, like dotted fields in Agda. These are strict hProps.

Current work: Miquel's implicit calculus of constructions (ICC), B. Barras and B. Bernardo's decidable version (ICC*)

Judgmental Equality

My Wishes: Reflection When We Can Have It

Limited Equality Reflection

I want equality reflection whenever it doesn't break things

$$(\forall\ (\mathtt{x} : \mathtt{T})\ (\mathtt{pf} : \mathtt{x} = \mathtt{x}),\ \mathtt{pf} = \mathtt{eq\_refl})$$
$$\to \forall\ (\mathtt{x} : \mathtt{T})\ (\mathtt{pf} : \mathtt{x} = \mathtt{x}),\ \mathtt{pf} \equiv \mathtt{eq\_refl}$$

Judgmental Equality

My Wishes: Reflection When We Can Have It

Limited Equality Reflection

I want equality reflection whenever it doesn't break things

```
(∀ (x : T) (pf : x = x), pf = eq_refl)
→ ∀ (x : T) (pf : x = x), pf ≡ eq_refl
```

(What's a general rule? Inductive type families with one constructor which are all provably equal to that constructor?)

Judgmental Equality

My Wishes: Postulating Judgmental Equality

Postulating Judgmental Equality?

Voevodsky suggests (and Dan Grayson has worked on implementing) having two equality types, a non-fibrant reflected equality type, and a fibrant intensional equality type. Perhaps Coq should go this route one day?

Judgmental Equality

My Wishes

I also want:

- $(\lambda\ \mathtt{x}\ \mathtt{y} \implies \mathtt{x} + \mathtt{y}) \equiv (\lambda\ \mathtt{x}\ \mathtt{y} \implies \mathtt{y} + \mathtt{x})$ (done in CoqMT by Pierre-Yves Strub)
- ability to add computation rules for axioms

Judgmental Equality

My Wishes

I also want:

- $(\lambda\ \mathtt{x}\ \mathtt{y} \Longrightarrow \mathtt{x} + \mathtt{y}) \equiv (\lambda\ \mathtt{x}\ \mathtt{y} \Longrightarrow \mathtt{y} + \mathtt{x})$ (done in CoqMT by Pierre-Yves Strub)
- ability to add computation rules for axioms
  - univalence

Judgmental Equality

My Wishes

I also want:

- $(\lambda\ \mathtt{x}\ \mathtt{y} \implies \mathtt{x} + \mathtt{y}) \equiv (\lambda\ \mathtt{x}\ \mathtt{y} \implies \mathtt{y} + \mathtt{x})$ (done in CoqMT by Pierre-Yves Strub)
- ability to add computation rules for axioms
  - univalence
  - functional extensionality

Judgmental Equality

My Wishes

I also want:

- `(λ x y ⟹ x + y)` $\equiv$ `(λ x y ⟹ y + x)` (done in CoqMT by Pierre-Yves Strub)
- ability to add computation rules for axioms
  - univalence
  - functional extensionality
  - higher inductive types

Judgmental Equality

My Wishes

I also want:

- `(`λ` x y ⟹ x + y) ≡ (`λ` x y ⟹ y + x)` (done in CoqMT by Pierre-Yves Strub)
- ability to add computation rules for axioms
  - univalence
  - functional extensionality
  - higher inductive types
  - internalized parametricity

Judgmental Equality

Implementation Properties

- should be optional extensions

Judgmental Equality

Implementation Properties

- should be optional extensions
- should be customizable, with plug-ins or flags or both

Judgmental Equality

Implementation Properties

- should be optional extensions
- should be customizable, with plug-ins or flags or both
- type-checking should still be decidable

Judgmental Equality

My Wishes: Why?

Why?

Judgmental Equality

My Wishes: Why?

Why?

Theorem proving is easier when the type-checker does more work for me.

Judgmental Equality

My Wishes: Why?

Why?

Theorem proving is easier when the type-checker does more work for me.

And it seems like an interesting system to play with.

Higher Inductive Types

Higher inductive types are:

Higher Inductive Types

Higher inductive types are:

- Inductive types

Higher Inductive Types

Higher inductive types are:

- Inductive types
- freely generated with higher path structure (non-trivial equalities)

Higher Inductive Types

Higher inductive types are:

- Inductive types
- freely generated with higher path structure (non-trivial equalities)

Example: The interval ($0 \rightsquigarrow 1$)

Higher Inductive Types

Higher inductive types are:

- Inductive types
- freely generated with higher path structure (non-trivial equalities)

Example: The interval ($0 \rightsquigarrow 1$)

```
Inductive Interval :=
| zero : Interval
| one  : Interval
| seg  : zero = one.
```

Higher Inductive Types

Why?

Higher inductive types are useful for:

Higher Inductive Types

Why?

Higher inductive types are useful for:

- Homotopy type theory (making basic spaces)

Higher Inductive Types

Why?

Higher inductive types are useful for:

- Homotopy type theory (making basic spaces)
- Quotient types

Higher Inductive Types

Why?

Higher inductive types are useful for:

- Homotopy type theory (making basic spaces)
- Quotient types
- Formalizing version control systems (according to Dan Licata[2])

[2] "Git as a HIT", http://dlicata.web.wesleyan.edu/pubs/l13git/git.pdf

Higher Inductive Types

Why?

Higher inductive types are useful for:

- Homotopy type theory (making basic spaces)
- Quotient types
- Formalizing version control systems (according to Dan Licata[2])
- Proving functional extensionality

[2] "Git as a HIT", http://dlicata.web.wesleyan.edu/pubs/l13git/git.pdf

Higher Inductive Types

Proving functional extensionality

```
Definition functional_extensionality A B f g
    : (∀ x, f x = g x) → f = g
  := λ H ⇒ f_equal
          (λ i x ⇒
            match i return B with
              | zero ⇒ f x
              | one  ⇒ g x
              | seg  ⇒ H x
            end)
          seg.
```

Higher Inductive Types

Proving functional extensionality

```
:= match seg in (_ = y)
     return ((λ x ⇒ f x)
               = (λ x ⇒ match y with
                          | zero ⇒ f x
                          | one  ⇒ g x
                          | seg  ⇒ H x
                        end))
   with
     | eq_refl => eq_refl
   end.
```

Higher Inductive Types

How?

Note that higher inductive types don't magically give you computational functional extensionality.

Higher Inductive Types

How?

Note that higher inductive types don't magically give you computational functional extensionality.

You must solve computational functional extensionality to implement computational HITs.

Higher Inductive Types

How?

Note that higher inductive types don't magically give you computational functional extensionality.

You must solve computational functional extensionality to implement computational HITs.

(Similar story for implementing computational univalence, another feature on my wishlist.)

Higher Inductive Types

How?

Note that higher inductive types don't magically give you computational functional extensionality.

You must solve computational functional extensionality to implement computational HITs.

(Similar story for implementing computational univalence, another feature on my wishlist.)

Breaks canonicity

Higher Inductive Types

How?

Note that higher inductive types don't magically give you computational functional extensionality.

You must solve computational functional extensionality to implement computational HITs.

(Similar story for implementing computational univalence, another feature on my wishlist.)

Breaks canonicity (jugdmentally),

Higher Inductive Types

How?

Note that higher inductive types don't magically give you computational functional extensionality.

You must solve computational functional extensionality to implement computational HITs.

(Similar story for implementing computational univalence, another feature on my wishlist.)

Breaks canonicity (jugdmentally), preserves it up to propositional equality? (conjecture by Voevodsky for UA)

Higher Inductive Types

Current Work

- Yves Bertot's private inductive types;[3] adapted by Matthieu Sozeau

[3] http://coq.inria.fr/files/coq5_submission_3.pdf

Higher Inductive Types

Current Work

- Yves Bertot's private inductive types;[3] adapted by Matthieu Sozeau
    - Comparatively easy to implement

[3] http://coq.inria.fr/files/coq5_submission_3.pdf

Higher Inductive Types

Current Work

- Yves Bertot's private inductive types;[3] adapted by Matthieu Sozeau
  - Comparatively easy to implement
  - Allows one to disable pattern matching on inductive types outside a module, which is sufficient to implement a trick by Dan Licata[4]

[3] http://coq.inria.fr/files/coq5_submission_3.pdf

[4] http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/

Higher Inductive Types

Current Work

- Yves Bertot's private inductive types;[3] adapted by Matthieu Sozeau
  - Comparatively easy to implement
  - Allows one to disable pattern matching on inductive types outside a module, which is sufficient to implement a trick by Dan Licata[4]
  - Equalities are axioms; not computational

[3] `http://coq.inria.fr/files/coq5_submission_3.pdf`

[4] `http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/`

Higher Inductive Types

Current Work

- Yves Bertot's private inductive types;[3] adapted by Matthieu Sozeau
  - Comparatively easy to implement
  - Allows one to disable pattern matching on inductive types outside a module, which is sufficient to implement a trick by Dan Licata[4]
  - Equalities are axioms; not computational
  - Only eliminators, no pattern matching

[3] http://coq.inria.fr/files/coq5_submission_3.pdf

[4] http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/

Higher Inductive Types

Current Work

- Yves Bertot's private inductive types;[3] adapted by Matthieu Sozeau
    - Comparatively easy to implement
    - Allows one to disable pattern matching on inductive types outside a module, which is sufficient to implement a trick by Dan Licata[4]
    - Equalities are axioms; not computational
    - Only eliminators, no pattern matching
- Burno Barras has some partial work that's more computational[5]

[3] http://coq.inria.fr/files/coq5_submission_3.pdf

[4] http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/

[5] https://github.com/barras/coq/tree/hit

Higher Inductive Types

My Wishes

I want:

Higher Inductive Types

My Wishes

I want:

- to be able to define and pattern match on higher inductive types

Higher Inductive Types

My Wishes

I want:

- to be able to define and pattern match on higher inductive types
- all tactics should support HITs

Higher Inductive Types

My Wishes

I want:

- to be able to define and pattern match on higher inductive types
- all tactics should support HITs
- judgmental reduction rules for matching on paths from HITs

Higher Inductive Types

My Wishes

I want:

- to be able to define and pattern match on higher inductive types
- all tactics should support HITs
- judgmental reduction rules for matching on paths from HITs
- equality should not be special

Higher Inductive Types

My Wishes

I want:

- to be able to define and pattern match on higher inductive types
- all tactics should support HITs
- judgmental reduction rules for matching on paths from HITs
- equality should not be special
  - typechecker should not depend on standard library

Higher Inductive Types

My Wishes

I want:

- to be able to define and pattern match on higher inductive types
- all tactics should support HITs
- judgmental reduction rules for matching on paths from HITs
- equality should not be special
  - typechecker should not depend on standard library
  - c.f. proposal for pattern matching justifying K[6]

[6] "The K axiom in Coq (almost) for free"
http://coq.inria.fr/files/adt-2fev10-corbineau.pdf

Higher Inductive Types (without equality in the kernel)

Possible Generalization (I)

- If equality isn't special, then HITs can put inhabitants in arbitrary types

Higher Inductive Types (without equality in the kernel)

Possible Generalization (I)

- If equality isn't special, then HITs can put inhabitants in arbitrary types
- BAD, if it allows us to give a proof of `False`

Higher Inductive Types (without equality in the kernel)

Possible Generalization (I)

- If equality isn't special, then HITs can put inhabitants in arbitrary types
- BAD, if it allows us to give a proof of `False`

```
Inductive BAD : Set :=
| silly : BAD
| terrible : False.
```

Higher Inductive Types (without equality in the kernel)

Possible Generalization (I)

- If equality isn't special, then HITs can put inhabitants in arbitrary types
- BAD, if it allows us to give a proof of `False`
- Idea: Require providing an inhabitant of the appropriate type family

Higher Inductive Types (without equality in the kernel)

Possible Generalization (I)

- If equality isn't special, then HITs can put inhabitants in arbitrary types
- BAD, if it allows us to give a proof of `False`
- Idea: Require providing an inhabitant of the appropriate type family
  - Used to pick out which branch of pattern matching to use

Higher Inductive Types (without equality in the kernel)

Possible Generalization (I)

- If equality isn't special, then HITs can put inhabitants in arbitrary types
- BAD, if it allows us to give a proof of `False`
- Idea: Require providing an inhabitant of the appropriate type family
  - Used to pick out which branch of pattern matching to use
  - Simply reduces when the provided term sits in the right type (not just right type family)

Higher Inductive Types (without equality in the kernel)

Possible Generalization (I)

```
Inductive Interval : Type :=
| zero : Interval
| one : Interval
| seg : zero = one
and picking
| seg : zero = _ := eq_refl.
```

Higher Inductive Types (without equality in the kernel)

Possible Generalization (II)

```
Inductive _==_ ‘(x : A) : ∀ {B}, B → Type :=
| refl1 : x == x
| refl2 : x == x.
Inductive foo : Type :=
| bar : nat → foo
| proof1 : ∀ (n : ℕ), bar 2 == bar (S (S n))
| proof2 : ∀ (n : ℕ), bar 0 == bar 1
and picking
| proof1 : ∀ n, bar 2 == _ := λ n ⇒ refl1
| proof2 : ∀ n, bar 0 == _ := λ n ⇒ refl2.
```

Higher Inductive Types (without equality in the kernel)

Possible Generalization (III)

Mike Shulman tells me this might be saying that a generalized higher inductive type is a polynomial functor F together with an object of $F(1)$.

Higher Inductive Types (without equality in the kernel)

Possible Generalization (III)

Mike Shulman tells me this might be saying that a generalized higher inductive type is a polynomial functor F together with an object of $F(1)$.

We still need computation rules for this.

Higher Inductive Types (without equality in the kernel)

Possible Generalization (III)

Mike Shulman tells me this might be saying that a generalized higher inductive type is a polynomial functor F together with an object of $F(1)$.

We still need computation rules for this. (See Appendix)

Higher Inductive Types (without equality in the kernel)

Possible Generalization (III)

Mike Shulman tells me this might be saying that a generalized higher inductive type is a polynomial functor F together with an object of $F(1)$.

We still need computation rules for this. (See Appendix)

Also an implementation, and justification of consistency.

The Rest of my Wishlist (I)

This was just a small (but important) part of my wishlist. The rest:

- a better story for namespacing[7]
- induction-recursion, induction-induction, etc.
- very dependent types, insanely dependent types (Σ as Π)[8]
- better coinduction (should be compositional, maybe based on copatterns)
- size/type-based termination
- support for explicit universe level variables (without loosing the default of typical ambiguity)

[7] `https://coq.inria.fr/bugs/show_bug.cgi?id=3171`

[8] `https://github.com/UlfNorell/insane`, "Formal Objects in Type Theory Using Very Dependent Types" `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.4169&rep=rep1&type=pdf`

The Rest of my Wishlist (II)

- parallel version of `all: solve` when there are no evars in the goal
- a search that searches the entire standard library, and not just currently `Required` files
- a search which is up to unification, rather than up to pattern matching
- coercions that don't care about the uniform inheritance condition[9]
- faster rewrite
- automatic generation of the equivalence between record types and nested sigma types
- ability to write theorems that apply to all records, which are specialized at type-inference time (a la typeclasses or mtac)

[9] `https://coq.inria.fr/bugs/show_bug.cgi?id=3115`

The Rest of my Wishlist (II)

- notations should be able to pick a meaning based on the type of their constituents (but must have a consistent scope for each term across all meanings) (can currently be hacked with boilerplate, typeclasses, and `$(...)$` to remove the typeclasses)[10]
- better handling of open terms in Ltac, and support for recursing under binders in tactics (maybe fixed with new tactic engine?)[11]
- easier use of ML plugins (I don't want to have to recompile them myself)
- typed/monadic tactic language

[10] https://coq.inria.fr/bugs/show_bug.cgi?id=3090

[11] https://coq.inria.fr/bugs/show_bug.cgi?id=3106 and https://coq.inria.fr/bugs/show_bug.cgi?id=3102

The Rest of my Wishlist (III)

- more uniform support for canonical structures (like ssr has)
- support for reflective simplification (maybe a native reifier which runs at type inference time, and a special type in the stdlib or something for syntax)
- rewrite that alternates simpl and argument inference
- rewrite which matches the head by pattern matching and the rest by unification
- variant of `@?` patterns for [pattern]ing on things other than bound indices and parameters, heuristically[12]
- have a `function_scope` like `type_scope`[13]

[12] https://coq.inria.fr/bugs/show_bug.cgi?id=3148
[13] https://coq.inria.fr/bugs/show_bug.cgi?id=3080

The Rest of my Wishlist (IV)

- a variant of `Hint Rewrite` which infers arguments based on pattern matching then runs `simpl` on the hypothesis, then rewrites with the simplified hypothesis
- 'where' clauses in records should permit abbreviations[14]
- variant of `abstract` which finishes the subproof with `Defined` rather than `Qed` (and another variant which finishes it with `Defined` and then runs `Global Opaque` on the constant)
- allow overriding symmetry, reflexivity[15]

[14] https://coq.inria.fr/bugs/show_bug.cgi?id=3066
[15] https://coq.inria.fr/bugs/show_bug.cgi?id=3113

The Rest of my Wishlist (V)

- etransitivity should take an optional term with holes[16]
- where clauses in records should support (`only parsing`)[17]
- support for simultaneous generation of terms binding scopes[18]
- better handling (speed-wise) of large terms and types (native projections might fix this)

[16] https://coq.inria.fr/bugs/show_bug.cgi?id=3065
[17] https://coq.inria.fr/bugs/show_bug.cgi?id=3067
[18] https://coq.inria.fr/bugs/show_bug.cgi?id=3123

Thanks!

Questions?

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

I want `match`es to distribute over arrows

```
match p as p' in (T x _)
  return (∀ y : T', T'' x p' y)
with
  | con1 ⇒ f1
  ...
end
≡
```

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

I want `matches` to distribute over arrows

```
≡ (λ y : T' ⇒
      match p as p' in (T x _)
        return (T'' x p' y)
      with
        | con1 ⇒ f1 y
        ...
      end)
```

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

I want a `match` whose branches unify to disappear (if the return type is constant)

```
match p return T with
  | _ ⇒ val
end ≡ val
```

Judgmental Equality

My Wishes: Computation Rules for match

More computation rules for match

I want matches to distribute over inductive types (when the branches unify appropriately)

```
match p as p' in (T x _)
  return (T' (f x p'))
with
  | con1 ⇒ Build_T' _ con1 val1
  ...
end
```

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

I want `match`es to distribute over inductive types (when the branches unify appropriately)

```
≡
Build_T'
   (match p with | con1 ⇒ f _ con1 | ... end
   (match p with | con1 ⇒ con1 | ... end)
   (match p with | con1 ⇒ val1 | ... end)
```

Judgmental Equality

My Wishes: Computation Rules for `match`

More computation rules for `match`

I want `match`es on `match`es to reduce to `match`es which return `match`es

```
match (match ... with ... end) with ... ⇒
≡
match ... with ... ⇒ ... (match ... with .
```

Computation Rules for HITs

Proposed computation rule for HITs

Given a higher inductive type `T` and a path constructor `p : a = b`, we should have

```
match p in (_ = y)
  return (P (fixmatch {h} y with
               | a => c
               | b => d
               | p => f
             end)) with
  | eq_refl => g
end
```

Computation Rules for HITs

Proposed computation rule for HITs

Given a higher inductive type `T` and a path constructor `p : a = b`, we should have

$\equiv$

```
match f in (_ = y) return (P y) with
  | eq_refl => g
end
```