Binary parametricity in Rocq

The case of record types

Vojtěch Štěpančík

Outline

Motivation

- Automation for large libraries of mathematics

Motivation

- Automation for large libraries of mathematics
- Codegen for types of morphisms

Motivation

- Automation for large libraries of mathematics
- Codegen for types of morphisms

Turn

```
Record Monoid A
  := {
    e : A;
    * : A → A → A;
    // axioms
  }.
```

Motivation

- Automation for large libraries of mathematics
- Codegen for types of morphisms

Turn

```
Record Monoid A
  := {
    e : A;
    * : A → A → A;
    // axioms
  }.
```

into

```
Record MonoidHom M M' (f : M → M')
  := {
    eR : ∀ a, f M.e = M'.e;
    mR : ∀ a b,
      f (a M.* b) = (f a) M'.* (f b);
  }.
```

Motivation

- Automation for large libraries of mathematics
- Codegen for types of morphisms

Turn

```
Record Monoid A
  := {
    e : A;
    * : A → A → A;
    // axioms
  }.
```

into

```
Record MonoidHom M M' (f : M → M')
  := {
    eR : ∀ a, f M.e = M'.e;
    mR : ∀ a b,
      f (a M.* b) = (f a) M'.* (f b);
  }.
```

- Maximal generality — parametricity translation

Parametricity translation

- Traditionally: take one interpretation, derive a relational interpretation

Parametricity translation

- Traditionally: take one interpretation, derive a relational interpretation
- With dependent types, relations are in the language:
  `Rel A B := (A → B → 𝒰)`

Parametricity translation

- Traditionally: take one interpretation, derive a relational interpretation
- With dependent types, relations are in the language: `Rel A B := (A → B → 𝒰)`
- Parametricity as syntactic translation/metaprogram `⟦-⟧`

Parametricity translation

- Traditionally: take one interpretation, derive a relational interpretation
- With dependent types, relations are in the language:
  `Rel A B := (A → B → 𝒰)`
- Parametricity as syntactic translation/metaprogram `⟦-⟧`
- Types translated to relations on their elements, e.g.

```
⟦𝒰⟧ := λ A B → (A → B → 𝒰)
⟦Π(x : A). P x⟧ := λ f f' → Π(a : A)(a' : A').
                           Π(aR : ⟦A⟧ a a'). ⟦P⟧ (f a) (f' a')
```

Parametricity translation

- Traditionally: take one interpretation, derive a relational interpretation
- With dependent types, relations are in the language:
  `Rel A B := (A → B → 𝒰)`
- Parametricity as syntactic translation/metaprogram `[[-]]`
- Types translated to relations on their elements, e.g.

```
[[𝒰]] := λ A B → (A → B → 𝒰)
[[Π(x : A). P x]] := λ f f' → Π(a : A)(a' : A').
                       Π(aR : [[A]] a a'). [[P]] (f a) (f' a')
```

- Elements translated to witnesses of these relations:
  `Γ ⊢ a : A  ⇒  [[Γ]] ⊢ [[a]] : [[A]] t t'`

Record types

- Idea: Type formers `K : Π(A : 𝒰). 𝒰` translated to `Π(A A' : 𝒰)(AR : A → A' → 𝒰). K A → K A' → 𝒰`

Record types

- Idea: Type formers `K : Π(A : 𝒰). 𝒰` translated to `Π(A A' : 𝒰)(AR : A → A' → 𝒰). K A → K A' → 𝒰`
- Instantiate `AR a a' := (f a = a')` for some `f : A → A'`

Record types

- Idea: Type formers `K : Π(A : 𝒰). 𝒰` translated to `Π(A A' : 𝒰)(AR : A → A' → 𝒰). K A → K A' → 𝒰`
- Instantiate `AR a a' := (f a = a')` for some `f : A → A'`
- Records are second-class, treated as inductives

Record types

- Idea: Type formers `K : Π(A : 𝒰). 𝒰` translated to `Π(A A' : 𝒰)(AR : A → A' → 𝒰). K A → K A' → 𝒰`
- Instantiate `AR a a' := (f a = a')` for some `f : A → A'`
- Records are second-class, treated as inductives

```
Record Monoid A
  := {
    e : A;
    * : A → A → A;
  }.
```

Record types

- Idea: Type formers `K : Π(A : 𝒰). 𝒰` translated to `Π(A A' : 𝒰)(AR : A → A' → 𝒰). K A → K A' → 𝒰`
- Instantiate `AR a a' := (f a = a')` for some `f : A → A'`
- Records are second-class, treated as inductives

```
Record Monoid A
  := {
    e : A;
    * : A → A → A;
  }.
```

```
Inductive Monoid A : Set :=
| build :
  A →
  (A → A → A) →
  Monoid A.
```

Record types

- Idea: Type formers `K : Π(A : 𝒰). 𝒰` translated to `Π(A A' : 𝒰)(AR : A → A' → 𝒰). K A → K A' → 𝒰`
- Instantiate `AR a a' := (f a = a')` for some `f : A → A'`
- Records are second-class, treated as inductives

```
Record Monoid A
  := {
    e : A;
    * : A → A → A;
  }.
```

```
Inductive Monoid A : Set :=
| build :
  A →
  (A → A → A) →
  Monoid A.
```

```
Inductive MonoidR A A' AR : Monoid A → Monoid A' → Set :=
| buildR : ∀ e e' (eR : AR e e') * *' (*R : ...),
           MonoidR (build e *) (build e' *')
```

Record types

```
Inductive MonoidR A A' AR : Monoid A → Monoid A' → Set :=
| buildR : ∀ e e' (eR : AR e e') * *' (*R : ...),
           MonoidR (build e *) (build e' *')
```

- `MonoidR` is indexed, no corresponding record!

Record types

```
Inductive MonoidR A A' AR : Monoid A → Monoid A' → Set :=
| buildR : ∀ e e' (eR : AR e e') * *' (*R : ...),
           MonoidR (build e *) (build e' *')
```

- `MonoidR` is indexed, no corresponding record!
- Bespoke treatment needed

rocq-elpi

- "Embeddable λProlog Interpreter" — Prolog with first-class functions

rocq-elpi

- "Embeddable λProlog Interpreter" — Prolog with first-class functions
- Nice HOAS API for Rocq

rocq-elpi

- "Embeddable λProlog Interpreter" — Prolog with first-class functions
- Nice HOAS API for Rocq
- Contains a base implementation of binary parametricity

rocq-elpi

- "Embeddable λProlog Interpreter" — Prolog with first-class functions
- Nice HOAS API for Rocq
- Contains a base implementation of binary parametricity
- Naïve implementation "works"

rocq-elpi

- "Embeddable λProlog Interpreter" — Prolog with first-class functions
- Nice HOAS API for Rocq
- Contains a base implementation of binary parametricity
- Naïve implementation "works"

```
pred param-record-decl i:record-decl, i:string, i:list (option constant),
  i:list term, i:list term, i:term, i:term, o:record-decl.
param-record-decl end-record _ _ _ _ _ _ end-record.
param-record-decl (field Attrs Id Ty Rest) Suffix [some Proj|Projs]
  RevParams1 RevParams2 X X' (field Attrs IdR TyRsubst RestR) :-
  IdR is (Id ^ Suffix),
  param Ty _Ty' TyR,
  coq.mk-app (global (const Proj)) {std.rev [X|RevParams1]} Proj1,
  coq.mk-app (global (const Proj)) {std.rev [X'|RevParams2]} Proj2,
  coq.subst-fun [Proj1, Proj2] TyR TyRsubst,
  @pi-parameter IdR TyRsubst fld\
    param Proj1 Proj2 fld ⟹
    param-record-decl (Rest Proj1) Suffix Projs RevParams1 RevParams2 X X' (RestR fld).
```

Problem

- Naïve implementation "works"

```
Record MonoidR A A' AR (M : Monoid A) (M' : Monoid A')
  := {
    eR : AR M.e M'.e;
    *R : ∀ a a' (aR : AR a a') b b' (bR : AR b b'),
         AR (M.* a b) (M'.* a' b');
  }.
```

Problem

- Naïve implementation "works"

```
Record MonoidR A A' AR (M : Monoid A) (M' : Monoid A')
  := {
    eR : AR M.e M'.e;
    *R : ∀ a a' (aR : AR a a') b b' (bR : AR b b'),
         AR (M.* a b) (M'.* a' b');
  }.
```

- "Constructors relate constructors" doesn't work, wrong types:

```
[[Π(A : 𝒰). A → (A → A → A) → Monoid A]] :=
Π(A A' AR e e' eR * *' *R).
MonoidR A A' AR (e, *) (e', *')
≠ Π(A A' AR M M' eR *R). MonoidR A A' AR M M'
```

Problem

- Naïve implementation "works"

```
Record MonoidR A A' AR (M : Monoid A) (M' : Monoid A')
  := {
    eR : AR M.e M'.e;
    *R : ∀ a a' (aR : AR a a') b b' (bR : AR b b'),
         AR (M.* a b) (M'.* a' b');
  }.
```

- "Constructors relate constructors" doesn't work, wrong types:

```
[[Π(A : 𝒰). A → (A → A → A) → Monoid A]] :=
Π(A A' AR e e' eR * *' *R).
MonoidR A A' AR (e, *) (e', *')
≠ Π(A A' AR M M' eR *R). MonoidR A A' AR M M'
```

- Fixed with a "compatibility constructor"

Conclusion

- Supported translations:
  - Simple cases like `Monoid`
  - Dependent fields
  - Primitive projections
- Next steps
  - Annotations à la Trocq
  - Removing irrelevant fields
  - Upstreaming