

Frame type theory

Cyril Cohen¹, Assia Mahboubi², Xavier Montillet²

¹Université Côte d'Azur, Inria, France

²Inria, LS2N, France

June 13, 2019*

Goal

We want a nice dependently-typed calculus that allows for modular definitions and proofs and minimizes boilerplate

Alternance of axioms and definitions: the advent of boilerplate

It's **not** about expressivity

```
Axiom car : Type
```

```
Definition binary_op := car → car → car
```

```
Axiom mult : binary_op (* written . *)
```

```
Definition mult_associative :  $\Pi$  (x,y,z : car),  
    (x · y) · z = x · (y · z)
```

```
Axiom op_associator : op_associative
```

Alternance of axioms and definitions: the advent of boilerplate

It's **not** about expressivity

```
Axiom car : Type
```

```
Definition binary_op := car → car → car
```

```
Axiom mult : binary_op (* written . *)
```

```
Definition mult_associative :  $\Pi$  (x,y,z : car),  
  (x · y) · z = x · (y · z)
```

```
Axiom op_associator : op_associative
```

```
Definition binary_op (car : Type) := car → car → car
```

```
Definition mult_associative (cat : Type)  
  (mult : binary_op car) :  $\Pi$  (x,y,z : car),  
  (x · y) · z = x · (y · z)
```

```
Axiom car : Type
```

```
Axiom mult : binary_op car (* written . *)
```

```
Axiom op_associator : op_associative car mult
```

Alternance of axioms and definitions: the advent of boilerplate

Too much boilerplate!

Definition def1 := ...

Definition def2 (proof1 : axiom1 def1) := ...

Definition def3 (proof1 : axiom1 def1)
 (proof2 : axiom2 def1 (def2 proof1)) := ...

Alternance of axioms and definitions: the advent of boilerplate

Too much boilerplate!

```
Definition def1 := ...
```

```
Definition def2 (proof1 : axiom1 def1) := ...
```

```
Definition def3 (proof1 : axiom1 def1)  
                (proof2 : axiom2 def1 (def2 proof1)) := ...
```

Problem

If n alternations, $\Theta(n^3)$ boilerplate code!

First-class objects, subsuming records

Definition by induction

```
1 Definition trivial_monoid : Mon :=
2   { car := unit, e := (), ... }
3 Definition monoid_product (M1 M2 : Mon) : Mon :=
4   { car := M1.car × M2.car, e := (M1.e, M2.e), ... }
```

First-class objects, subsuming records

Definition by induction

```
1 Definition trivial_monoid : Mon :=
2   { car := unit, e := (), ... }
3 Definition monoid_product (M1 M2 : Mon) : Mon :=
4   { car := M1.car × M2.car, e := (M1.e, M2.e), ... }

```

```
5 Definition monoid_power (M : Mon) (n : nat) : Mon := {
6   car := iter unit (λx. x × M.car) n,
7   e := iter () (λx. (x, M.car)),
8   ...
9 }
```

First-class objects, subsuming records

Definition by induction

```
1 Definition trivial_monoid : Mon :=
2   { car := unit, e := (), ... }
3 Definition monoid_product (M1 M2 : Mon) : Mon :=
4   { car := M1.car × M2.car, e := (M1.e, M2.e), ... }

```

```
5 Definition monoid_power (M : Mon) (n : nat) : Mon := {
6   car := iter unit (λx. x × M.car) n,
7   e := iter () (λx. (x, M.car)),
8   ...
9 }

```

```
10 Definition monoid_power (M : Mon) (n : nat) : Mon :=
11   iter unit_monoid (λx. monoid_product x M)

```

First-class objects, subsuming records

Quantification

```
1 Lemma quantifier_elimination :  
2    $\Pi$  (F : ReadClosedField),  
3    $\Pi$  ( $\varphi$  : Formula),  
4    $\Sigma$  ( $\psi$  : ClosedFormula),  
5    $\llbracket \varphi \rrbracket_F = \llbracket \psi \rrbracket_F$ 
```

Named abstraction and application

- ▶ No α -renaming: Abstraction replaced by empty fields:
 $\lambda x_1.(x_2 := t, \lambda x_3.(x_4 := t', x_5 := t''))$ becomes
 $x_1 :=?, x_2 := t, x_3 :=?, x_4 := t', x_5 := t''$

Named abstraction and application

- ▶ No α -renaming: Abstraction replaced by empty fields:

$\lambda x_1.(x_2 := t, \lambda x_3.(x_4 := t', x_5 := t''))$ becomes

$x_1 := ?, x_2 := t, x_3 := ?, x_4 := t', x_5 := t''$

- ▶ Reduction:

$\{(x_1 := ?, x_2 := ?, x_3 := x_1 + x_2, x_4 := ?, x_5 := x_3 + x_4) (x_1 := 2, x_2 := 3)\} .x_3$

∇_*

$\{x_1 := 2, x_2 := 3, x_3 := x_1 + x_2, x_4 := ?, x_5 := x_3 + x_4\} .x_3$

∇_*

$\{x_1 := 2, x_2 := 3, x_3 := 2 + 3, x_4 := ?, x_5 := x_3 + x_4\} .x_3$

∇_*

$2 + 3$

Definitional singleton, subsuming modules

- ▶ Type may need to know the value of a field:

$$\{x^{\text{Type}} := \mathbb{N}, y^x := 0, z^{y=0} := \text{refl}\} : \Sigma x^{\text{Type}}, \Sigma y^x, \Sigma z^{y=0}$$

Definitional singleton, subsuming modules

- ▶ Type may need to know the value of a field:

$$\{x^{\text{Type}} := \mathbb{N}, y^x := 0, z^{y=0} := \text{refl}\} : \cancel{\Sigma x^{\text{Type}}, \Sigma y^x, \Sigma z^{y=0}}$$

Definitional singleton, subsuming modules

- ▶ Type may need to know the value of a field:

$$\{x^{\text{Type}} := \mathbb{N}, y^x := 0, z^{y=0} := \text{refl}\} : \cancel{\Sigma x^{\text{Type}}, \Sigma y^x, \Sigma z^{y=0}}$$

- ▶ Remember the value in the type;

$$\{x^{\text{Type}} \stackrel{\Delta}{=} \mathbb{N}, y^x \stackrel{\Sigma}{=} 0, z^{y=0} \stackrel{\Sigma}{=} \text{refl}\} : \Delta x^{\text{Type}} := \mathbb{N}, \Sigma y^x, \Sigma z^{y=0}$$

Definitional singleton, subsuming modules

- ▶ Type may need to know the value of a field:

$$\{x^{\text{Type}} := \mathbb{N}, y^x := 0, z^{y=0} := \text{refl}\} : \Sigma x^{\text{Type}}, \Sigma y^x, \Sigma z^{y=0}$$

- ▶ Remember the value in the type;

$$\{x^{\text{Type}} \stackrel{\Delta}{=} \mathbb{N}, y^x \stackrel{\Sigma}{=} 0, z^{y=0} \stackrel{\Sigma}{=} \text{refl}\} : \Delta x^{\text{Type}} := \mathbb{N}, \Sigma y^x, \Sigma z^{y=0}$$

- ▶ Typing rules

$$\frac{\Gamma \vdash \varphi : \Sigma x^A, \mathcal{B}}{\Gamma \vdash \varphi.x : A}$$

$$\frac{\Gamma \vdash \varphi : \Delta x^A := t, \mathcal{B}}{\Gamma \vdash \varphi.x : A}$$

$$\frac{\Gamma \vdash \varphi : \Delta x^A := t, \mathcal{B}}{\Gamma \vdash \varphi.x \equiv t : A}$$

Definitional singleton, subsuming modules

- ▶ Type may need to know the value of a field:

$$\{x^{\text{Type}} := \mathbb{N}, y^x := 0, z^{y=0} := \text{refl}\} : \cancel{\Sigma x^{\text{Type}}, \Sigma y^x, \Sigma z^{y=0}}$$

- ▶ Remember the value in the type;

$$\{x^{\text{Type}} \stackrel{\Delta}{=} \mathbb{N}, y^x \stackrel{\Sigma}{=} 0, z^{y=0} \stackrel{\Sigma}{=} \text{refl}\} : \Delta x^{\text{Type}} := \mathbb{N}, \Sigma y^x, \Sigma z^{y=0}$$

- ▶ Typing rules

$$\frac{\Gamma \vdash \varphi : \Sigma x^A, \mathcal{B}}{\Gamma \vdash \varphi.x : A} \quad \frac{\Gamma \vdash \varphi : \Delta x^A := t, \mathcal{B}}{\Gamma \vdash \varphi.x : A} \quad \frac{\Gamma \vdash \varphi : \Delta x^A := t, \mathcal{B}}{\Gamma \vdash \varphi.x \equiv t : A}$$

- ▶ Simulates let expressions:

$$\text{let } x := t \text{ in } u \approx \{x \stackrel{\Delta}{=} t, y := u\}.y$$

Field commutations, maybe subsuming sections

If $x \notin \text{FV}(u) \cap \text{FV}(B)$ and $y \notin \text{FV}(t) \cap \text{FV}(A)$, then:

$$\begin{aligned}\Sigma x^A, \Sigma y^B, \mathcal{C} &\equiv \Sigma y^B, \Sigma x^A, \mathcal{C} \\ \Delta x^A := t, \Delta y^B := u, \mathcal{C} &\equiv \Delta y^B := u, \Delta x^A := t, \mathcal{C} \\ \Delta x^A := t, \Sigma y^B, \mathcal{C} &\equiv \Sigma y^B, \Delta x^A := t, \mathcal{C}\end{aligned}$$

$$x^A := t, y^B := u, \varphi \equiv y^B := u, x^A := t, \varphi$$

$$\Pi x^A, \Pi y^B, \mathcal{C} \equiv \Pi y^B, \Pi x^A, \mathcal{C}$$

$$x^A := ?, y^B := ?, \varphi \equiv y^B := ?, x^A := ?, \varphi$$

Field commutations, maybe subsuming sections

If $x \notin \text{FV}(u) \cap \text{FV}(B)$ and $y \notin \text{FV}(t) \cap \text{FV}(A)$, then:

$$\begin{aligned}\Sigma x^A, \Sigma y^B, \mathcal{C} &\equiv \Sigma y^B, \Sigma x^A, \mathcal{C} \\ \Delta x^A := t, \Delta y^B := u, \mathcal{C} &\equiv \Delta y^B := u, \Delta x^A := t, \mathcal{C} \\ \Delta x^A := t, \Sigma y^B, \mathcal{C} &\equiv \Sigma y^B, \Delta x^A := t, \mathcal{C}\end{aligned}$$

$$x^A := t, y^B := u, \varphi \equiv y^B := u, x^A := t, \varphi$$

$$\Pi x^A, \Pi y^B, \mathcal{C} \equiv \Pi y^B, \Pi x^A, \mathcal{C}$$

$$x^A := ?, y^B := ?, \varphi \equiv y^B := ?, x^A := ?, \varphi$$

Need more commutations to simulate minimal discharge of sections

Modularity

- ▶ Renaming operator: necessary to combine libraries with different naming conventions
- ▶ Includes:

$$\begin{aligned} \text{PtTopSpace} &:= \{ \text{car}^{\text{Type}} :=?, \text{pt}^{\text{car}} :=?, \text{top}^{\mathcal{P}(\mathcal{P}(\text{car}))} :=?, \dots \} \\ \text{Group} &:= \{ \text{car}^{\text{Type}} :=?, \text{e}^{\text{car}} :=?, \text{mult}^{\text{car} \rightarrow \text{car} \rightarrow \text{car}} :=?, \dots \} \\ \text{TopGroup} &:= \{ \\ &\quad \text{include}(\text{Group}), \\ &\quad \text{smart_include}(\text{rename}_{\text{pt} \rightarrow \text{e}}(\text{Group})), \\ &\quad \text{mult_cont} := \dots, \\ &\quad \dots \\ &\} \end{aligned}$$

Conclusion

- ▶ Work in progress: candidate calculus
- ▶ Expected properties:
 - ▶ Conservative over MLTT (with definitional singleton)
 - ▶ Subsumes modules (minus subtyping) and records (and hopefully section)
 - ▶ No code duplication, and minimal boilerplate