Frame type theory

Cyril Cohen[1], Assia Mahboubi[2], Xavier Montillet[2]

[1]Université Côte d'Azur, Inria, France
[2]Inria, LS2N, France

June 13, 2019*

---

*Last updated on June 14, 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 : Π (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 : Π (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) : Π (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

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

First-class objects, subsuming records

Definition by induction

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

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

First-class objects, subsuming records

Definition by induction

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

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

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

First-class objects, subsuming records

Quantification

```
Lemma quantifier_elimination :
  Π (F : ReadClosedField),
  Π (φ : Formula),
  Σ (ψ : ClosedFormula),
  ⟦φ⟧_F = ⟦ψ⟧_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:

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

$$\triangledown_*$$

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

$$\triangledown_*$$

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

$$\triangledown_*$$

$$2 + 3$$

Definitional singleton, subsuming modules

- Type may need to know the value of a field:
  $\left\{x^{\mathrm{Type}} := \mathbb{N}, y^{x} := 0, z^{y=0} := \mathrm{refl}\right\} : \Sigma x^{\mathrm{Type}}, \Sigma y^{x}, \Sigma z^{y=0}$

Definitional singleton, subsuming modules

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

  $$\left\{x^{\mathrm{Type}} := \mathbb{N}, y^{x} := 0, z^{y=0} := \mathrm{refl}\right\} : \xcancel{\Sigma x^{\mathrm{Type}}, \Sigma y^{x}, \Sigma z^{y=0}}$$

Definitional singleton, subsuming modules

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

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

- Remember the value in the type;

  $$\left\{x^{\mathrm{Type}} \overset{\Delta}{:=} \mathbb{N}, y^{x} \overset{\Sigma}{:=} 0, z^{y=0} \overset{\Sigma}{:=} \mathrm{refl}\right\} : \Delta x^{\mathrm{Type}} := \mathbb{N}, \Sigma y^{x}, \Sigma z^{y=0}$$

Definitional singleton, subsuming modules

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

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

- Remember the value in the type;

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

- Typing rules

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

Definitional singleton, subsuming modules

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

$$\left\{x^{\mathrm{Type}} := \mathbb{N}, y^{x} := 0, z^{y=0} := \mathrm{refl}\right\} : \xcancel{\Sigma x^{\mathrm{Type}}, \Sigma y^{x}, \Sigma z^{y=0}}$$

- Remember the value in the type;

$$\left\{x^{\mathrm{Type}} \overset{\Delta}{:=} \mathbb{N}, y^{x} \overset{\Sigma}{:=} 0, z^{y=0} \overset{\Sigma}{:=} \mathrm{refl}\right\} : \Delta x^{\mathrm{Type}} := \mathbb{N}, \Sigma y^{x}, \Sigma z^{y=0}$$

- Typing rules

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

- Simulates let expressions:

$$\mathrm{let}\, x := t\, \mathrm{in}\, u \approx \left\{x \overset{\Delta}{:=} t, y := u\right\}.y$$

Field commutations, maybe subsuming sections

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

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

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

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

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

Field commutations, maybe subsuming sections

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

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

$$\begin{aligned}
\Pi x^A, \Pi y^B, \mathcal{C} &\equiv \Pi y^B, \Pi x^A, \mathcal{C} \\
\\
x^A :=?, y^B :=?, \varphi &\equiv y^B :=?, x^A :=?, \varphi
\end{aligned}$$

Need more commutations to simulate minimal discharge of sections

Modularity

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

$$
\begin{aligned}
&\mathrm{PtTopSpace} := \left\{ \mathit{car}^{\mathrm{Type}} :=?, \mathit{pt}^{\mathit{car}} :=?, \mathit{top}^{\mathcal{P}(\mathcal{P}(\mathit{car}))} :=?, \dots \right\} \\
&\mathrm{Group} := \left\{ \mathit{car}^{\mathrm{Type}} :=?, \mathit{e}^{\mathit{car}} :=?, \mathit{mult}^{\mathit{car} \to \mathit{car} \to \mathit{car}} :=?, \dots \right\} \\
&\mathrm{TopGroup} := \{ \\
&\quad \mathrm{include}\,(\mathrm{Group})\,, \\
&\quad \mathrm{smart\_include}\,(\mathrm{rename}_{\mathrm{pt} \mapsto \mathrm{e}}\,(\mathrm{Group}))\,, \\
&\quad \mathit{mult\_cont} := \dots, \\
&\quad \dots \\
&\}
\end{aligned}
$$

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