Automated Construction of Examples in Algebraic Geometry

Jesse Vogel

Leiden University

21 July, 2022
Introduction

Topics in Algebraic Geometry:

making algebraic geometry more concrete

Goal: to create a searchable database of properties, theorems and examples in algebraic geometry, and answer questions like:

- ‘Does there exist a scheme with property A but not property B?’
- ‘If a morphism has property A, does it also have property B?’
Introduction

Topics in Algebraic Geometry:

*making algebraic geometry more concrete*

**Goal**: to create a searchable database of properties, theorems and examples in algebraic geometry, and answer questions like:

- ‘Does there exist a scheme with property A but not property B?’
- ‘If a morphism has property A, does it also have property B?’
Initial (naive) idea

- database = lists of properties, examples and theorems
- property = string
- example = list of booleans
- theorem = list of assumptions + list of conclusions
- searching = iterating + blindly applying theorems
What about?

- composition of morphisms
- fiber products of schemes
- other categories: rings, modules, topological spaces, ...
- functors: Spec, $\Gamma$, Hom, forget, ...
- families of examples: $\mathbb{A}^1_X$ for any $X$
What about?

- composition of morphisms
- fiber products of schemes
- other categories: rings, modules, topological spaces, ...
- functors: Spec, $\Gamma$, Hom, forget, ...
- families of examples: $\mathbb{A}^1_X$ for any $X$
link to the new version of website
- Dependent type system (written in C++)

- By default, $\text{Prop} : \text{Type}$ and $\text{Type} : \text{Type}$

- Everything is a function

```javascript
Function {
    name: String
    type: Function
    parameters: List Function
}
```
Dependent type system (written in C++)

- By default, Prop : Type and Type : Type

- Everything is a function

```cpp
Function {
    name: String
    type: Function
    parameters: List Function
}
```
- Dependent type system (written in C++)

- By default $\text{Prop : Type}$ and $\text{Type : Type}$

- Everything is a function

```plaintext
Function {
    name: String
    type: Function
    parameters: List Function
}
```
- Dependent type system (written in C++)

- By default \( \text{Prop} : \text{Type} \) and \( \text{Type} : \text{Type} \)

- Everything is a function

Function {
    name: String
    type: Function
    parameters: List Function
}
Examples

let Ring : Type

let domain (R : Ring) : Prop

let domain_of_field {R : Ring} (h : field R) : domain R

let affine_line (X : Scheme) : Scheme

let zariski_local (P (X : Scheme) : Prop) : Prop
Some functions are specializations

Specialization extends Function {
    base: Function
    arguments: List Function
}

For example

let f (a b c : A) : B
def g (x : A) := f x x x

Can construct expressions, axioms, theorems, examples, ...

Theorems and examples are represented by the same type of object
Some functions are specializations

Specialization extends Function {
    base: Function
    arguments: List Function
}

For example

```latex
let f (a b c : A) : B
def g (x : A) := f x x x
```

Can construct expressions, axioms, theorems, examples, ...

Theorems and examples are represented by the same type of object
Some functions are specializations

Specialization extends Function {
    base: Function
    arguments: List Function
}

For example

```plaintext
let f (a b c : A) : B
def g (x : A) := f x x x
```

Can construct expressions, axioms, theorems, examples, ...

Theorems and examples are represented by the same type of object
Some functions are specializations

Specialization extends Function {
    base: Function
    arguments: List Function
}

For example

let f (a b c : A) : B
def g (x : A) := f x x x

Can construct expressions, axioms, theorems, examples, ...
search \( (X : \text{Scheme}) \ (h1 : \text{integral} \ X) \ (h2 : \text{affine} \ X) \)

1. Create ‘query’ (telescope)
2. Resolve goals (starting at the back), by applying functions, creating new queries
3. Continue recursively (breadth-first search) with some maximal depth, creating a tree of queries
4. When we reach an empty query, do backsubstitution
\textbf{Canard}

\begin{equation}
\text{search } (X : \text{Scheme}) \ (h1 : \text{integral} \ X) \ (h2 : \text{affine} \ X)
\end{equation}

1. Create ‘query’ (telescope)

2. Resolve goals (starting at the back), by applying functions, creating new queries

3. Continue recursively (breadth-first search) with some maximal depth, creating a tree of queries

4. When we reach an empty query, do backsubstitution
search (X : Scheme) (h1 : integral X) (h2 : affine X)

(1) Create ‘query’ (telescope)

(2) Resolve goals (starting at the back), by applying functions, creating new queries

(3) Continue recursively (breadth-first search) with some maximal depth, creating a tree of queries

(4) When we reach an empty query, do back substitution
search (X : Scheme) (h1 : integral X) (h2 : affine X)

(1) Create ‘query’ (telescope)
(2) Resolve goals (starting at the back), by applying functions, creating new queries
(3) Continue recursively (breadth-first search) with some maximal depth, creating a tree of queries
(4) When we reach an empty query, do backsubstitution
search \((X : \text{Scheme}) \ (h1 : \text{integral} \ X) \ (h2 : \text{affine} \ X)\)

1. Create ‘query’ (telescope)
2. Resolve goals (starting at the back), by applying functions, creating new queries
3. Continue recursively (breadth-first search) with some maximal depth, creating a tree of queries
4. When we reach an empty query, do backsubstitution
-- (1) Start with this query

search (X : Scheme) (h1 : integral X) (h2 : affine X)

-- (2) Apply this theorem

spec_af (R : Ring) : affine (Spec R)

-- to get this query (remember X := Spec R, h2 := spec_af R)

search (R : Ring) (h1 : integral (Spec R))

-- (3) Apply this theorem

spec_int {R : Ring} (h : domain R) : integral (Spec R)

-- to get this query (remember h1 := spec_int h)

search (R : Ring) (h : domain R)

-- (4) Apply this theorem

ZZ_is_dm : domain ZZ

-- and done! (with R := ZZ and h := ZZ_is_dm)
(1) Start with this query
search (X : Scheme) (h1 : integral X) (h2 : affine X)
(2) Apply this theorem
spec_af (R : Ring) : affine (Spec R)
to get this query (remember X := Spec R, h2 := spec_af R)
search (R : Ring) (h1 : integral (Spec R))
(3) Apply this theorem
spec_int {R : Ring} (h : domain R) : integral (Spec R)
to get this query (remember h1 := spec_int h)
search (R : Ring) (h : domain R)
(4) Apply this theorem
ZZ_is_dm : domain ZZ
and done! (with R := ZZ and h := ZZ_is_dm)
Canard

-- (1) Start with this query
search (X : Scheme) (h1 : integral X) (h2 : affine X)

-- (2) Apply this theorem
spec_af (R : Ring) : affine (Spec R)

-- to get this query (remember X := Spec R, h2 := spec_af R)
search (R : Ring) (h1 : integral (Spec R))

-- (3) Apply this theorem
spec_int {R : Ring} (h : domain R) : integral (Spec R)

-- to get this query (remember h1 := spec_int h)
search (R : Ring) (h : domain R)

-- (4) Apply this theorem
ZZ_is_dm : domain ZZ

-- and done! (with R := ZZ and h := ZZ_is_dm)
-- (1) Start with this query
search (X : Scheme) (h1 : integral X) (h2 : affine X)

-- (2) Apply this theorem
spec_af (R : Ring) : affine (Spec R)
-- to get this query (remember X := Spec R, h2 := spec_af R)
search (R : Ring) (h1 : integral (Spec R))

-- (3) Apply this theorem
spec_int {R : Ring} (h : domain R) : integral (Spec R)
-- to get this query (remember h1 := spec_int h)
search (R : Ring) (h : domain R)

-- (4) Apply this theorem
ZZ_is_dm : domain ZZ
-- and done! (with R := ZZ and h := ZZ_is_dm)
Note

- If goal has arguments, create ‘local context’ with variables
  
  \[ \text{search} \ (P \ (X \ : \ \text{Scheme}) \ : \ \text{Prop}) \]

Optimizations

- Sort functions based on type before-hand
- Prioritize based on depth
- Cut-off unnecessary branches
- Multi-threading
Note

- If goal has arguments, create ‘local context’ with variables
  
  \texttt{search (P (X : Scheme) : Prop)}

Optimizations

- Sort functions based on type before-hand
- Prioritize based on depth
- Cut-off unnecessary branches
- Multi-threading
Now on to Lean(4)!
-- (1) Mark functions with an attribute
@[aesop safe] theorem my_thm (P : Prop) : P := by { ... }
@[aesop unsafe 42%] axiom my_ax (R : Ring) : trivial R

-- (2) Use Aesop as tactic
theorem my_awesome_thm (R : Ring) : reduced R := by {
    aesop;
}

-- (3) possibly with extra theorems
example (R : Ring) : reduced R := by {
    aesop (add unsafe 10% my_awesome_thm);
}
Aesop (by Jannis Limperg)

-- (1) Mark functions with an attribute
@[aesop safe] theorem my_thm (P : Prop) : P := by { ... }
@[aesop unsafe 42%] axiom my_ax (R : Ring) : trivial R

-- (2) Use Aesop as tactic
theorem my_awesome_thm (R : Ring) : reduced R := by {
    aesop;
}

-- (3) possibly with extra theorems
example (R : Ring) : reduced R := by {
    aesop (add unsafe 10% my_awesome_thm);
}
-- (1) Mark functions with an attribute
@[aesop safe] theorem my_thm (P : Prop) : P := by { ... }
@[aesop unsafe 42%] axiom my_ax (R : Ring) : trivial R

-- (2) Use Aesop as tactic
theorem my_awesome_thm (R : Ring) : reduced R := by {
    aesop;
}

-- (3) possibly with extra theorems
example (R : Ring) : reduced R := by {
    aesop (add unsafe 10% my_awesome_thm);
}
-- (1) Mark functions with an attribute
@[aesop safe] theorem my_thm (P : Prop) : P := by { ... }
@[aesop unsafe 42%] axiom my_ax (R : Ring) : trivial R

-- (2) Use Aesop as tactic
theorem my_awesome_thm (R : Ring) : reduced R := by {
  aesop;
}

-- (3) possibly with extra theorems
example (R : Ring) : reduced R := by {
  aesop (add unsafe 10% my_awesome_thm);
}
Aesop

How does Aesop search?

(1) apply safe rules

(2) apply unsafe rules, prioritize based on percentages
#query command

\[
\texttt{#query } (X : \text{Scheme}) \ (h : X.\text{affine}) : (q : X.\text{quasi\_compact})
\]

\[
\forall (X : \text{Scheme}) \ (h : X.\text{affine}), \exists (q : X.\text{quasi\_compact}), \text{True}
\]

call Aesop

extract objects and proofs

pretty print
- Fix bugs / test cases
- User-friendly interface (website)
- Enlarge database
- Integrate with mathlib / swap definitions