A Coq Library for Finite Types

1^{st} bachelor seminar talk

Jan Menz

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Advisor: Prof. Dr. rer. nat. Gert Smolka

13. Juni 2016

Contents

Finite types

What is a finite type?

Finite types

What is a finite type?

- Type

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

What do we need formally?

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

What do we need formally?

- Type

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

What do we need formally?

- Type
- List of inhabitants

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

What do we need formally?

- Type
- List of inhabitants
- Completeness proof for list

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

What do we need formally?

- Type
- List of inhabitants
- Completeness proof for list
- Decidability of equality

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

What do we need formally?

- Type
- List of inhabitants
- Completeness proof for list
- Decidability of equality
  - needed for completeness proof

My goal

Finite types (sets) are mathematically uninteresting

My goal

Finite types (sets) are mathematically uninteresting

- Well understood

My goal

Finite types (sets) are mathematically uninteresting

- Well understood
- No big surprises

My goal

Finite types (sets) are mathematically uninteresting

- Well understood
- No big surprises
- Easy to use

My goal

Finite types (sets) are mathematically uninteresting

- Well understood
- No big surprises
- Easy to use
- **This is their advantage**

My goal

Finite types (sets) are mathematically uninteresting

- Well understood
- No big surprises
- Easy to use
- **This is their advantage**

My goal

Finite types (sets) are mathematically uninteresting

- Well understood
- No big surprises
- Easy to use
- **This is their advantage**

Challenge: make them uninteresting in type theory

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

What do we need formally?

- Type
- List of inhabitants
- Completeness proof for list
- Decidability of equality

Realisitation in Coq

Reminder: eqType

```
Definition dec (P: ℙ) := {P} + {¬P}
Notation "eq_dec X" :=
(∀ x y: X, dec (x = y))(at level 70)
Structure eqType := EqType {
 eqtype :> Type ;
 decide_eq : eq_dec eqtype }.
```

Realisitation in Coq

First idea:

```
Structure finType: Type := FinType {
  type : eqType;
  elements: list type;
  allIn: ∀ x: type, count elements x = 1
 }.
```

Reminder: eqType

```
Definition dec (P: ℙ) := {P} + {¬P}
Notation "eq_dec X" :=
(∀ x y: X, dec (x = y)) (at level 70)
Structure eqType := EqType {
  eqtype :> Type ;
  decide_eq : eq_dec eqtype }.
```

Realisitation in Coq

First idea:

```
Structure finType: Type := FinType {
 type : eqType;
 elements: list type;
 allIn: ∀ x: type, count elements x = 1
}.
```

count

$$
\begin{aligned}
&count\ []\ x && = 0 \\
&count\ (x :: A)\ x && = 1 + count\ A\ x \\
&count\ (y :: A)\ x && = count\ A\ x && x \neq y
\end{aligned}
$$

Realisitation in Coq

First idea:

```
Structure finType: Type := FinType {
  type : eqType;
  elements: list type;
  allIn: ∀ x: type, count elements x = 1
}.
```

count

$$
\begin{aligned}
&count\ []\ x && = 0 \\
&count\ (x :: A)\ x && = 1 + count\ A\ x \\
&count\ (y :: A)\ x && = count\ A\ x && x \neq y
\end{aligned}
$$

We want to use it like the “real” type

Realisitation in Coq

First idea:

```
Structure finType: Type := FinType {
  type :> eqType;
  elements: list type;
  allIn: ∀ x: type, count elements x = 1
}.
```

count

$$\begin{aligned} count\ []\ x &= 0 \\ count\ (x :: A)\ x &= 1 + count\ A\ x \\ count\ (y :: A)\ x &= count\ A\ x && x \neq y \end{aligned}$$

We want to use it like the “real” type

Type classes

- Define class of types as type class

Reminder: Decidability

```
Existing Class dec.
```

Type classes

- Define class of types as type class
- For type in this class: Define an instance

Reminder: Decidability

```
Existing Class dec.
Instance bool_eq_dec:
eq_dec 𝔹.
```

Type classes

- Define class of types as type class
- For type in this class: Define an instance
- Instance is used, when element of this type need to be inferred

Reminder: Decidability

```
Existing Class dec.
Instance bool_eq_dec:
eq_dec 𝔹.
Definition EqBool := EqType 𝔹
```

Type classes

- Define class of types as type class
- For type in this class: Define an instance
- Instance is used, when element of this type need to be inferred

Reminder: Decidability

```
Existing Class dec.
Instance bool_eq_dec:
eq_dec 𝔹.
Definition EqBool := EqType 𝔹
```

Only one Instance for each type

Type Classes

Make finType dependent on types:

```
Class finTypeC (type: eqType): Type := FinTypeC {
  elements: list type;
  allIn: ∀ x: type, count elements x = 1
}.
```

Type Classes

Make finType dependent on types:

```
Class finTypeC (type: eqType): Type := FinTypeC {
  elements: list type;
  allIn: ∀ x: type, count elements x = 1
}.

Structure finType: Type := FinType {
 type :> eqType;
 class : finTypeC type }.
```

Type classes

Nice:

Type classes

Nice:
finTypes/eqTypes can be automatically generated from types

Type classes

<u>Nice:</u>

finTypes/eqTypes can be automatically generated from types

```
Definition toeqType (T: Type ) {e: eq_dec T}:
eqType := EqType T.
```

Type classes

Nice:
finTypes/eqTypes can be automatically generated from types

```
Definition toeqType (T: Type ) {e: eq_dec T}:
eqType := EqType T.
```

Problematic:

Type classes

Nice:

finTypes/eqTypes can be automatically generated from types

```
Definition toeqType (T: Type ) {e: eq_dec T}:
eqType := EqType T.
```

Problematic:

finTypes/eqTypes cannot be inferred from elements of the type:

Type classes

<u>Nice:</u>
finTypes/eqTypes can be automatically generated from types

```
Definition toeqType (T: Type ) {e: eq_dec T}:
eqType := EqType T.
```

<u>Problematic:</u>
finTypes/eqTypes cannot be inferred from elements of the type:

```
Compute (count [true;false] true).
```

Type classes

Nice:
finTypes/eqTypes can be automatically generated from types

```
Definition toeqType (T: Type ) {e: eq_dec T}:
eqType := EqType T.
```

Problematic:
finTypes/eqTypes cannot be inferred from elements of the type:

```
Compute (count [true;false] true).
```

```
Error: (diff)
The term "[true; false]" has type "list bool"
while it is expected to have type "list ?X".
```

Canonical Structures

- Extend Coqs unification algoritm

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures
- Every time they syntactically “fit” they are inserted

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures
- Every time they syntactically "fit" they are inserted
- Can be combined to powerful *telescopes*

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures
- Every time they syntactically “fit” they are inserted
- Can be combined to powerful *telescopes*

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures
- Every time they syntactically “fit” they are inserted
- Can be combined to powerful *telescopes*

```
Canonical Structure EqBool := EqType 𝔹.
```

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures
- Every time they syntactically “fit” they are inserted
- Can be combined to powerful *telescopes*

```
Canonical Structure EqBool := EqType 𝔹.
Canonical Structure finType_bool := FinType EqBool.
```

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures
- Every time they syntactically “fit” they are inserted
- Can be combined to powerful *telescopes*

```
Canonical Structure EqBool := EqType 𝔹.
Canonical Structure finType_bool := FinType EqBool.
Compute (count [true;false] true).
```

Canonical Structures

- Extend Coqs unification algoritm
- Arbitrary values can be declared as canonical structures
- Every time they syntactically “fit” they are inserted
- Can be combined to powerful *telescopes*

```
Canonical Structure EqBool := EqType 𝔹.
Canonical Structure finType_bool := FinType EqBool.
Compute (count [true;false] true).
= if bool_eq_dec true true
then S (if bool_eq_dec true false then 1 else 0)
else if bool_eq_dec true false then 1 else 0 : ℕ
```

Together: Powerful inference

```
Definition finType_BoolUnit := tofinType(𝔹 × unit).
finType_BoolUnit is defined
```

What does this actually look like?

Together: Powerful inference

```
Definition finType_BoolUnit := tofinType(𝔹 × unit).
finType_BoolUnit is defined
```

What does this actually look like?

```
finType_BoolUnit = @tofinType (𝔹 × unit)
(@decide_eq (EqCross EqBool EqUnit))
(finTypeC_Cross finType_bool finType_unit)
: finType
```

Together: Powerful inference

```
Definition finType_BoolUnit := tofinType(𝔹 × unit).
finType_BoolUnit is defined
```

What does this actually look like?

```
finType_BoolUnit = @tofinType (𝔹 × unit)
(@decide_eq (EqCross EqBool EqUnit))
(finTypeC_Cross finType_bool finType_unit)
: finType
```

inferred with canonical structures

Together: Powerful inference

```
Definition finType_BoolUnit := tofinType(𝔹 × unit).
finType_BoolUnit is defined
```

What does this actually look like?

```
finType_BoolUnit = @tofinType (𝔹 × unit)
(@decide_eq (EqCross EqBool EqUnit))
(finTypeC_Cross finType_bool finType_unit)
: finType
```

inferred with canonical structures
inferred with type classes

Equivalence Principles

Finite Types satisfy important equivalences:

About: elem

elem is a projection from a finType to its list of elements

Equivalence Principles

Finite Types satisfy important equivalences:

$$(\forall\ (x : F), p\ x) \leftrightarrow \forall\ x \in (elem\ F), p\ x$$

About: elem

elem is a projection from a finType to its list of elements

Equivalence Principles

Finite Types satisfy important equivalences:

$$(\forall\ (x : F), p\ x) \leftrightarrow \forall\ x \in (elem\ F), p\ x$$
$$(\exists\ (x : F), p\ x) \leftrightarrow \exists\ x \in (elem\ F), p\ x$$

About: elem

elem is a projection from a finType to its list of elements

Equivalence Principles

Finite Types satisfy important equivalences:

$$(\forall\ (x : F), p\ x) \leftrightarrow \forall\ x \in (elem\ F), p\ x$$
$$(\exists\ (x : F), p\ x) \leftrightarrow \exists\ x \in (elem\ F), p\ x$$
$$(\exists\ (x : F), p\ x) \leftrightarrow \exists\ x, x \in (elem\ F) \rightarrow p\ x$$

About: elem

elem is a projection from a finType to its list of elements

Equivalence Principles

Finite Types satisfy important equivalences:

$$(\forall\ (x : F), p\ x) \leftrightarrow \forall\ x \in (elem\ F), p\ x$$
$$(\exists\ (x : F), p\ x) \leftrightarrow \exists\ x \in (elem\ F), p\ x$$
$$(\exists\ (x : F), p\ x) \leftrightarrow \exists\ x, x \in (elem\ F) \rightarrow p\ x$$

About: elem

elem is a projection from a finType to its list of elements

First one allows to use induction

Interesting equalities

finTypes are equal to "ordinary" types due to coercion

Interesting equalities

finTypes are equal to "ordinary" types due to coercion

$$\mathit{tofinType}\ X = X$$

About: (x) and ?

(x) and ? are my operators for the finType of a cartesian product of finTypes and an option type of a finType, respectively.

Interesting equalities

finTypes are equal to "ordinary" types due to coercion

$$\begin{aligned} \mathit{tofinType}\ X &= X \\ \mathbb{B} &= \mathit{finType\_bool} \end{aligned}$$

About: (x) and ?

(x) and ? are my operators for the finType of a cartesian product of finTypes and an option type of a finType, respectively.

Interesting equalities

finTypes are equal to "ordinary" types due to coercion

$$\begin{aligned} \mathit{tofinType}\ X &= X \\ \mathbb{B} &= \mathit{finType\_bool} \\ F_1 \times F_2 &= F_1\ (x)\ F_2 \end{aligned}$$

About: (x) and ?

(x) and ? are my operators for the finType of a cartesian product of finTypes and an option type of a finType, respectively.

Interesting equalities

finTypes are equal to "ordinary" types due to coercion

$$\begin{aligned} \mathit{tofinType}\ X &= X \\ \mathbb{B} &= \mathit{finType\_bool} \\ F_1 \times F_2 &= F_1\ (x)\ F_2 \\ \text{option}\ F &= ?\ F \end{aligned}$$

About: (x) and ?

(x) and ? are my operators for the finType of a cartesian product of finTypes and an option type of a finType, respectively.

Interesting equalities

finTypes are equal to "ordinary" types due to coercion

$$\begin{aligned} \mathit{tofinType}\ X &= X \\ \mathbb{B} &= \mathit{finType\_bool} \\ F_1 \times F_2 &= F_1\ (x)\ F_2 \\ \text{option}\ F &= ?\ F \\ \mathit{tofinType}\ \mathbb{B} &= \mathit{finType\_bool} \end{aligned}$$

About: (x) and ?

(x) and ? are my operators for the finType of a cartesian product of finTypes and an option type of a finType, respectively.

Interesting equalities

finTypes are equal to "ordinary" types due to coercion

$$
\begin{aligned}
\mathit{tofinType}\ X &= X \\
\mathbb{B} &= \mathit{finType\_bool} \\
F_1 \times F_2 &= F_1\ (x)\ F_2 \\
\text{option}\ F &= ?\ F \\
\mathit{tofinType}\ \mathbb{B} &= \mathit{finType\_bool} \\
\mathit{tofinType}(F_1 \times F_2) &= F_1\ (x)\ F_2
\end{aligned}
$$

About: (x) and ?

(x) and ? are my operators for the finType of a cartesian product of finTypes and an option type of a finType, respectively.

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs

EXTENSIONAL POWER (SET THEORETIC FUNCTIONS)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$
- STF is uniquely identified by its image as a list

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$
- STF is uniquely identified by its image as a list
  - $[false; true]$

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$
- STF is uniquely identified by its image as a list
  - $[false; true]$
- We can model the type of all STF ($F_1 \longrightarrow F_2$) as a finite type

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$
- STF is uniquely identified by its image as a list
  - $[false; true]$
- We can model the type of all STF ($F_1 \longrightarrow F_2$) as a finite type
  - bundle image and proof for correct length

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$
- STF is uniquely identified by its image as a list
  - $[false; true]$
- We can model the type of all STF ($F_1 \longrightarrow F_2$) as a finite type
  - bundle image and proof for correct length
  - ```
    Definition STF (F:finType) (X:Type ) :=
    {image: list X | if |image| = |X| then ⊤ else ⊥}
    ```
    ```

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$
- STF is uniquely identified by its image as a list
  - $[false; true]$
- We can model the type of all STF ($F_1 \longrightarrow F_2$) as a finite type
  - bundle image and proof for correct length
  - `Definition STF (F:finType) (X:Type ) := {image: list X | if |image| = |X| then ⊤ else ⊥}`
- *extensionalPower* function computes list of all STF

Extensional power (Set theoretic functions)

- Set theoretic functions (STF): sets of pairs
  - $neg := \{(true, false), (false, true)\}$
- (x:F) is uniquely identified by position in *elem*
  - $elem\ finType\_bool := [true; false]$
- STF is uniquely identified by its image as a list
  - $[false; true]$
- We can model the type of all STF ($F_1 \longrightarrow F_2$) as a finite type
  - bundle image and proof for correct length
  - ```
    Definition STF (F:finType) (X:Type ) :=
    {image: list X | if |image| = |X| then ⊤ else ⊥}
    ```
- *extensionalPower* function computes list of all STF
  - used in finType definition
    ```

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF
- $F_1 \longrightarrow F_2$ convertible to $F_1 \to F_2$

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF
- $F_1 \longrightarrow F_2$ convertible to $F_1 \to F_2$
  - applySTF

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF
- $F_1 \longrightarrow F_2$ convertible to $F_1 \to F_2$
  - applySTF
  - applySTF coercion to functions

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF
- $F_1 \longrightarrow F_2$ convertible to $F_1 \to F_2$
  - applySTF
  - applySTF coercion to functions
  - therefore STF usable as functions

FUNCTIONS AND STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF
- $F_1 \longrightarrow F_2$ convertible to $F_1 \to F_2$
  - applySTF
  - applySTF coercion to functions
  - therefore STF usable as functions
- (f: $F_1 \to F_2$) : $\forall x$, (toSTF f) x = f x

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF
- $F_1 \longrightarrow F_2$ convertible to $F_1 \to F_2$
  - applySTF
  - applySTF coercion to functions
  - therefore STF usable as functions
- (f: $F_1 \to F_2$) : $\forall x$, (toSTF f) x = f x
- (f: $F_1 \longrightarrow F_2$): toSTF f = f

Functions and STF

- $F_1 \to F_2$ convertible to $F_1 \longrightarrow F_2$
  - toSTF
- $F_1 \longrightarrow F_2$ convertible to $F_1 \to F_2$
  - applySTF
  - applySTF coercion to functions
  - therefore STF usable as functions
- (f: $F_1 \to F_2$) : $\forall x$, applySTF (toSTF f) x = f x
- (f: $F_1 \longrightarrow F_2$): toSTF (applySTF f) = f

Overview: Already done

- Formalisation of finite types

Overview: Already done

- Formalisation of finite types
- Basic types

Overview: Already done

- Formalisation of finite types
- Basic types
  - True

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties
  - option types

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties
  - option types
  - cartesian product

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties
  - option types
  - cartesian product
  - sum type

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties
  - option types
  - cartesian product
  - sum type
  - extensional power (set theoretic functions)

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties
  - option types
  - cartesian product
  - sum type
  - extensional power (set theoretic functions)
- Cardinality

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties
  - option types
  - cartesian product
  - sum type
  - extensional power (set theoretic functions)
- Cardinality
  - injective $(f : X \to Y) \to |X| \leq |Y|$

Overview: Already done

- Formalisation of finite types
- Basic types
  - True
  - False
  - unit
  - empty_Set
  - bool
- Closure properties
  - option types
  - cartesian product
  - sum type
  - extensional power (set theoretic functions)
- Cardinality
  - injective $(f : X \to Y) \to |X| \leq |Y|$
  - surjective $(f : X \to Y) \to |X| \geq |Y|$

Overview: still to do

- Order

OVERVIEW: STILL TO DO

- Order
- Choice

Overview: still to do

- Order
- Choice
- Closure properties

Overview: still to do

- Order
- Choice
- Closure properties
  - dependent pairs

Overview: still to do

- Order
- Choice
- Closure properties
  - dependent pairs
- Subtypes

Overview: still to do

- Order
- Choice
- Closure properties
  - dependent pairs
- Subtypes
- Fixed points

Overview: still to do

- Order
- Choice
- Closure properties
  - dependent pairs
- Subtypes
- Fixed points
- finType $\to$ countable type

Overview: still to do

- Order
- Choice
- Closure properties
  - dependent pairs
- Subtypes
- Fixed points
- finType $\to$ countable type
- *Possibly graphs*

Sources and Inspiration

Mahboubi, Assia and Tassi, Enrico
Canonical Structures for the working Coq user
ITP 2013, 4th Conference on Interactive Theorem Proving

Gonthier, Georges
ssreflect coqdoc documentation
http://math-comp.github.io/math-comp/htmldoc/index.html

Castéran, Pierre and Sozeau,Matthieu
A Gentle Introduction to Type Classes and Relations in Coq
http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typec

The End

Thank you for your attention

Any questions? Ask away!