A Coq Library for Finite Types

Bachelor Talk

Jan Menz

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Advisor: Prof. Dr. Gert Smolka

July 29, 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

Representation:

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X
- Duplicate free list of all inhabitants (*elem* X)

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X
- Duplicate free list of all inhabitants (*elem* X)
- Proof that list satisfies the properties

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X
- Duplicate free list of all inhabitants ($elem\ X$)
- Proof that list satisfies the properties
  - $\forall x, count\ (elem\ X)\ x = 1$

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X
- Duplicate free list of all inhabitants ($elem\ X$)
- Proof that list satisfies the properties
  - $\forall x, count\ (elem\ X)\ x = 1$

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X
- Duplicate free list of all inhabitants ($elem\ X$)
- Proof that list satisfies the properties
  - $\forall x, count\ (elem\ X)\ x = 1$

Cardinality

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X
- Duplicate free list of all inhabitants ($elem\ X$)
- Proof that list satisfies the properties
  - $\forall x, count\ (elem\ X)\ x = 1$

Cardinality

- Number of inhabitants of X

Finite types

What is a finite type?

- Type
- Finite number of inhabitants

Representation:

- Discrete Type X
- Duplicate free list of all inhabitants ($elem\ X$)
- Proof that list satisfies the properties
  - $\forall x, count\ (elem\ X)\ x = 1$

Cardinality

- Number of inhabitants of X
- $|elem\ X|$

Three usability features

If Y is a finite or discrete type, then

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$
  - Coercions

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$
  - Coercions
- Y can be automatically inferred.

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$
  - Coercions
- Y can be automatically inferred.
  - $\mathit{count}\ [\mathit{true}, \mathit{false}]\ \mathit{true} = 1$

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$
  - Coercions
- Y can be automatically inferred.
  - $\mathit{count}\ [\mathit{true}, \mathit{false}]\ \mathit{true} = 1$
  - Canonical structures

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$
  - Coercions
- Y can be automatically inferred.
  - $\mathit{count}\ [\mathit{true}, \mathit{false}]\ \mathit{true} = 1$
  - Canonical structures
- We can compute Y out of its base type.

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$
  - Coercions
- Y can be automatically inferred.
  - $\mathit{count}\ [\mathit{true}, \mathit{false}]\ \mathit{true} = 1$
  - Canonical structures
- We can compute Y out of its base type.
  - $\mathit{Cardinality}(\mathit{tofinType}\ \mathbb{B}) = 2$

Three usability features

If Y is a finite or discrete type, then

- Y can be used as a type.
  - $\forall (Y : \mathit{finType})(x : Y),\ x \in \mathit{elem}\ Y$
  - Coercions
- Y can be automatically inferred.
  - $\mathit{count}\ [\mathit{true}, \mathit{false}]\ \mathit{true} = 1$
  - Canonical structures
- We can compute Y out of its base type.
  - $\mathit{Cardinality}(\mathit{tofinType}\ \mathbb{B}) = 2$
  - Mainly type classes

Conversion to lists

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

Conversion to lists

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

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

Conversion to lists

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

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

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

Classical properties

Finite types often behave classically:

From list conversions:

Classical properties

Finite types often behave classically:

From list conversions:

Fact

For a decidable predicate p over X

- $\forall\, x : X,\ p\ x$ is decidable.

Classical properties

Finite types often behave classically:

From list conversions:

Fact

For a decidable predicate p over X

- $\forall\, x : X,\ p\ x$ is decidable.
- $\exists\, x : X,\ p\ x$ is decidable.

Classical properties

Finite types often behave classically:

From list conversions:

Fact

For a decidable predicate p over X

- $\forall\, x : X,\ p\ x$ is decidable.
- $\exists\, x : X,\ p\ x$ is decidable.
- $(\exists\, x : X,\ p\ x) \leftrightarrow \neg\, \forall\, x : X,\ (\neg\, p\ x)$.

Classical properties

Finite types often behave classically:

From list conversions:

Fact

For a decidable predicate p over X

- $\forall\, x : X,\ p\ x$ is decidable.
- $\exists\, x : X,\ p\ x$ is decidable.
- $(\exists\, x : X,\ p\ x) \leftrightarrow \neg\, \forall\, x : X,\ (\neg\, p\ x)$.
- $\neg\, (\forall\, x : X, p\ x) \leftrightarrow \exists\, x : X, \neg\, p\ x$.

Classical properties

Finite types often behave classically:

From list conversions:

Fact

For a decidable predicate p over X

- $\forall\, x : X,\ p\ x$ is decidable.
- $\exists\, x : X,\ p\ x$ is decidable.
- $(\exists\, x : X,\ p\ x) \leftrightarrow \neg\, \forall\, x : X,\ (\neg\, p\ x)$.
- $\neg\, (\forall\, x : X, p\ x) \leftrightarrow \exists\, x : X, \neg\, p\ x$.
- There is a constructive choice function $\exists\, x : X, p\ x \to \{x : X \mid p\ x\}$.

Cardinality

Fact

Let A be a list over X. Then

Cardinality

Fact

Let A be a list over X. Then

- $Cardinality\ X = card\ (elem\ X)$.

Cardinality

Fact

Let A be a list over X. Then

- $Cardinality\ X = card\ (elem\ X)$.
- $Cardinality\ X \geq card\ A$.

Cardinality

Fact

Let A be a list over X. Then

- $\mathit{Cardinality}\ X = \mathit{card}\ (\mathit{elem}\ X)$.
- $\mathit{Cardinality}\ X \geq \mathit{card}\ A$.
- $\mathit{Cardinality}\ X \geq |A|$, if A is duplicate free.

Pigeonhole principles

Pigeon hole principle from set theory also hold on finite types:

Pigeonhole principles

Pigeon hole principle from set theory also hold on finite types:

$(\exists \text{ injection } f : X_1 \to X_2) \to \mathit{Cardinality}\ X_1 \leq \mathit{Cardinality}\ X_2$

Pigeonhole principles

Pigeon hole principle from set theory also hold on finite types:

$(\exists \text{ injection } f : X_1 \to X_2) \to \mathit{Cardinality}\ X_1 \leq \mathit{Cardinality}\ X_2$

$(\exists \text{ surjection } f : X_1 \to X_2) \to \mathit{Cardinality}\ X_1 \geq \mathit{Cardinality}\ X_2$

Pigeonhole principles

Pigeon hole principle from set theory also hold on finite types:

$(\exists \text{ injection } f : X_1 \to X_2) \to \mathit{Cardinality}\ X_1 \leq \mathit{Cardinality}\ X_2$

$(\exists \text{ surjection } f : X_1 \to X_2) \to \mathit{Cardinality}\ X_1 \geq \mathit{Cardinality}\ X_2$

$(\exists \text{ bijection } f : X_1 \to X_2) \to \mathit{Cardinality}\ X_1 = \mathit{Cardinality}\ X_2$

Vectors

Usually: Collection of objects of some "type" with fixed size.

$$\mathbb{R}^n : \begin{pmatrix} \pi \\ e \\ \vdots \\ 0 \end{pmatrix} \begin{matrix} 1 \\ 2 \\ \vdots \\ n \end{matrix}$$

indexed by some number n

Vectors

Now: Collection of objects of some type Y with fixed size.

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \text{X-indexed Y vector} : \begin{pmatrix} y_1 & 1 \\ y_2 & 2 \\ \vdots & \vdots \\ y_n & n \end{pmatrix}$$

indexed by some finite type X

Vectors

Now: Collection of objects of some type Y with fixed size.

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \text{X-indexed Y vector} : \begin{pmatrix} y_1 & 1 \\ y_2 & 2 \\ \vdots & \vdots \\ y_n & n \end{pmatrix}$$

indexed by some finite type X

In Coq dependent pair: $\{ A \mid |A| = Cardinality\ X \}$

Vectors

Function interpretation:

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{matrix} \longrightarrow \\ \longrightarrow \\ \vdots \\ \longrightarrow \end{matrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} =: f$$

Vectors

Function interpretation:

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{matrix} \longrightarrow \\ \longrightarrow \\ \vdots \\ \longrightarrow \end{matrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} =: f$$

$X \longrightarrow Y := X$-indexed Y vector.

Vectors

Function interpretation:

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{matrix} \longrightarrow \\ \longrightarrow \\ \vdots \\ \longrightarrow \end{matrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} =: f$$

$$X \longrightarrow Y := X\text{-indexed } Y \text{ vector.}$$

$$image\ f := [y_1, y_2, \ldots, y_n]$$

Discreteness of vectors

We want to decide equality

In Coq dependent pair: $\{A \mid |A| = \mathit{Cardinality}\ X\}$

Discreteness of vectors

We want to decide equality

In Coq dependent pair: $\{A \mid |A| = \mathit{Cardinality}\ X\}$

Definition (Pure predicates[15])

A predicate $p : X \to \mathbb{P}$ is called *pure* if for every $x{:}X$ there is only one proof of $p\ x$.

Discreteness of vectors

We want to decide equality

In Coq dependent pair: $\{A \mid |A| = \mathit{Cardinality}\ X\}$

Definition (Pure predicates[15])

A predicate $p : X \to \mathbb{P}$ is called *pure* if for every $x{:}X$ there is only one proof of $p\ x$. Decidable predicates can be converted to pure predicates.

Discreteness of vectors

We want to decide equality

In Coq dependent pair: $\{A \mid \text{pure } (|A| = \mathit{Cardinality}\ X)\ \}$

Definition (Pure predicates[15])

A predicate $p : X \to \mathbb{P}$ is called *pure* if for every $x{:}X$ there is only one proof of $p\ x$. Decidable predicates can be converted to pure predicates.

Discreteness of vectors

We want to decide equality

In Coq dependent pair: $\{A \mid \text{pure } (|A| = \mathit{Cardinality}\ X)\}$

Definition (Pure predicates[15])

A predicate $p : X \to \mathbb{P}$ is called *pure* if for every $x{:}X$ there is only one proof of $p\ x$. Decidable predicates can be converted to pure predicates.

$\Rightarrow$ Vectors are discrete and extensional:

Discreteness of vectors

We want to decide equality

In Coq dependent pair: $\{A \mid \text{pure } (|A| = \mathit{Cardinality}\ X)\}$

Definition (Pure predicates[15])

A predicate $p : X \to \mathbb{P}$ is called *pure* if for every $x{:}X$ there is only one proof of $p\ x$. Decidable predicates can be converted to pure predicates.

$\Rightarrow$ Vectors are discrete and extensional:

Fact

Let f and g be two vectors. Then $\mathit{image}\ f = \mathit{image}\ g \to f = g$.

A finite vector type

Theorem

There is a finite type $X_2^{X_1}$ such that $X_2^{X_1} = X_1 \longrightarrow X_2$.

A finite vector type

Theorem

There is a finite type $X_2^{X_1}$ such that $X_2^{X_1} = X_1 \longrightarrow X_2$.

Theorem

$\mathit{Cardinality}\ X_2^{X_1} = (\mathit{Cardinality}\ X_2)^{\mathit{Cardinality}\ X_1}$.

Vectors vs. functions

Function interpretation:

f vector $X \longrightarrow Y$

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{matrix} \longrightarrow \\ \longrightarrow \\ \vdots \\ \longrightarrow \end{matrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} =: f$$

Vectors vs. functions

Function interpretation:

f vector $X \longrightarrow Y$

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{matrix} \longrightarrow \\ \longrightarrow \\ \vdots \\ \longrightarrow \end{matrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} =: f$$

Function *applyVect:* $X \longrightarrow Y \to X \to Y$.

VECTORS VS. FUNCTIONS

Function interpretation:

f vector $X \longrightarrow Y$

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{matrix} \longrightarrow \\ \longrightarrow \\ \vdots \\ \longrightarrow \end{matrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} =: f$$

Function *applyVect:* $X \longrightarrow Y \to X \to Y$.

Defined as coercion. We can write $f\ x$ instead of $applyVect\ f\ x$.

Vectors vs. functions

Vector interpretation:

f function $X \to Y$

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Vectors vs. functions

Vector interpretation:

f function $X \to Y$

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad vectorise\ f := \begin{pmatrix} f\ x_1 \\ f\ x_2 \\ \vdots \\ f\ x_n \end{pmatrix}$$

Vectors vs. functions

Vector interpretation:
f function $X \to Y$

$$elem\ X := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad vectorise\ f := \begin{pmatrix} f\ x_1 \\ f\ x_2 \\ \vdots \\ f\ x_n \end{pmatrix}$$

Function *vectorise:* $(X \to Y) \to (X \longrightarrow Y)$.

Vectors vs. functions

applyVect and *vectorise* are inverse functions:

Vectors vs. functions

applyVect and *vectorise* are inverse functions:

Theorem

Let f be a vector $X \longrightarrow Y$. Then *vectorise* $f = f$.

Vectors vs. functions

applyVect and *vectorise* are inverse functions:

Theorem

Let f be a vector $X \longrightarrow Y$. Then $\mathit{vectorise}\ f = f$.

Theorem

Let f be a function $X \to Y$. Then $(\mathit{vectorise}\ f)\ x = f\ x$.

More compount types

- Cartesian product

More compount types

- Cartesian product
- Sum

More compount types

- Cartesian product
- Sum
- Option

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality
- General dependent pairs

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality
- General dependent pairs
  - Uses Hedberg's theorem

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality
- General dependent pairs
  - Uses Hedberg's theorem
  - No equation for equality

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality
- General dependent pairs
  - Uses Hedberg's theorem
  - No equation for equality
- From a list

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality
- General dependent pairs
  - Uses Hedberg's theorem
  - No equation for equality
- From a list
  - Uses subtypes

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality
- General dependent pairs
  - Uses Hedberg's theorem
  - No equation for equality
- From a list
  - Uses subtypes
- Vectors

More compount types

- Cartesian product
- Sum
- Option
- Dependent pairs to $\mathbb{P}$ (subtypes)
  - Uses pure predicates
  - No equation for cardinality
- General dependent pairs
  - Uses Hedberg's theorem
  - No equation for equality
- From a list
  - Uses subtypes
- Vectors
  - Uses subtypes

Finite closure iteration[14, 13]

Compute subset of finite type F:

Finite closure iteration[14, 13]

Compute subset of finite type F:

- predicate *step*: $\mathit{list}\ X \to X \to \mathbb{P}$

Finite closure iteration[14, 13]

Compute subset of finite type F:

- predicate *step*: $list\ X \to X \to \mathbb{P}$
- function *pick*: $\forall\ A, \{x \mid step\ A\ x \wedge \neg\ (x \in A)\} + \forall\ x, step\ A\ x \to x \in A.$

FINITE CLOSURE ITERATION[14, 13]

Compute subset of finite type F:

- predicate *step*: $list\ X \to X \to \mathbb{P}$
- function *pick*: $\forall\ A, \{x \mid step\ A\ x \wedge \neg\ (x \in A)\} + \forall\ x, step\ A\ x \to x \in A.$

FCStep

```
Definition FCStep A :=
match (pick A) with
|inl L ⇒ match L with
          |exists _ x _ ⇒ x::A end
|inr _ ⇒ A end.
```

Finite closure iteration[14, 13]: Idea

Iterate *FCStep* until it reaches a fixed point

Finite closure iteration[14, 13]: Idea

Iterate *FCStep* until it reaches a fixed point

How many times?

Fact

Let A be a list over X and `FCIter := FCStep`$^{\texttt{Cardinality X}}$. Then `FCIter A` is a fixed point of *FCStep*.

FCIter induction [14, 13]

Induction principle for predicates preserved by *FCStep*:

FCIter induction [14, 13]

Induction principle for predicates preserved by *FCStep*:
$A \subseteq p := \forall\, x \in A, p\, x$

FCIter induction [14, 13]

Induction principle for predicates preserved by *FCStep*:
$A \subseteq p := \forall\, x \in A, p\ x$

Theorem

Let p be a predicate over X and A a list over X. Then
$A \subseteq p \to (\forall\, A\ x, A \subseteq p \to \mathit{step}\ A\ x \to p\ x) \to \mathit{FCIter}\ A \subseteq p.$

Least fixed points

Corollary

Let A be a list over X. Then `FCIter A` is a fixed point of *FCStep*.

Is it a least fixed point?

Least fixed points

Corollary

Let A be a list over X. Then `FCIter A` is a fixed point of *FCStep*.

Is it a least fixed point?

No! But ...

Least fixed points containing A

Definition (Least fixed points containing A)

Let A: *list* Y and f: *list* $Y \to$ *list* Y. A fixed point B of f is called the *least fixed point containing* A if $A \subseteq B$ and for any other fixed point B' of f: $A \subseteq B' \to B \subseteq B'$.

Least fixed points containing A

Definition (Least fixed points containing A)

Let A: *list* Y and f: *list* $Y \to$ *list* Y. A fixed point B of f is called the *least fixed point containing* A if $A \subseteq B$ and for any other fixed point B' of f: $A \subseteq B' \to B \subseteq B'$.

Definition (Consistency)

A *step* predicate is called *consistent* if
$\forall\, A\ x,\ step\ A\ x \to \forall\, A', A \subseteq A' \to step\ A'\ x$.

Least fixed points containing A

Definition (Least fixed points containing A)

Let A: *list* Y and f: *list* $Y \to$ *list* Y. A fixed point B of f is called the *least fixed point containing* A if $A \subseteq B$ and for any other fixed point B' of f: $A \subseteq B' \to B \subseteq B'$.

Definition (Consistency)

A *step* predicate is called *consistent* if
$\forall\, A\ x, \mathit{step}\ A\ x \to \forall\, A', A \subseteq A' \to \mathit{step}\ A'\ x$.

Theorem

If *step* is consistent then for any A the list *FCIter* A is the least fixed point containing A.

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

- A finite type S, the set of states.

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

- A finite type S, the set of states.
- Some s of type S, the start state.

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

- A finite type S, the set of states.
- Some s of type S, the start state.
- A decidable predicate F over S to define the accepting states.

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

- A finite type S, the set of states.
- Some s of type S, the start state.
- A decidable predicate F over S to define the accepting states.
- A transition function $\delta_S : S \to \Sigma \to S$.

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

- A finite type S, the set of states.
- Some s of type S, the start state.
- A decidable predicate F over S to define the accepting states.
- A transition function $\delta_S : S \to \Sigma \to S$.

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

- A finite type S, the set of states.
- Some s of type S, the start state.
- A decidable predicate F over S to define the accepting states.
- A transition function $\delta_S : S \to \Sigma \to S$.

We lift δ_S as δ_S^* to words.

Test case: Finite automata with finite types

Inspired by [3] (Talk on Monday 3:15 pm)

Assume a finite type Σ as the alphabet.

Deterministic finite automata are formalised by:

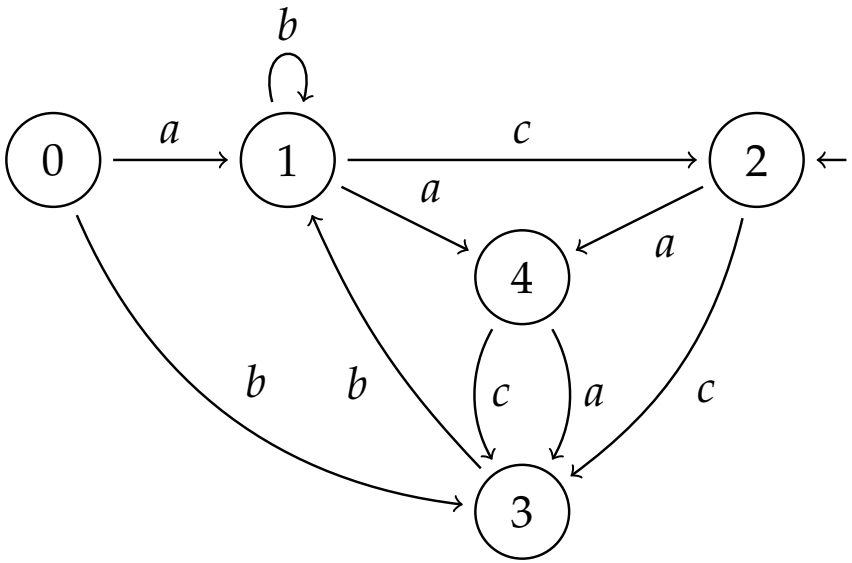
- A finite type S, the set of states.
- Some s of type S, the start state.
- A decidable predicate F over S to define the accepting states.
- A transition function $\delta_S : S \to \Sigma \to S$.

We lift δ_S as δ_S^* to words.

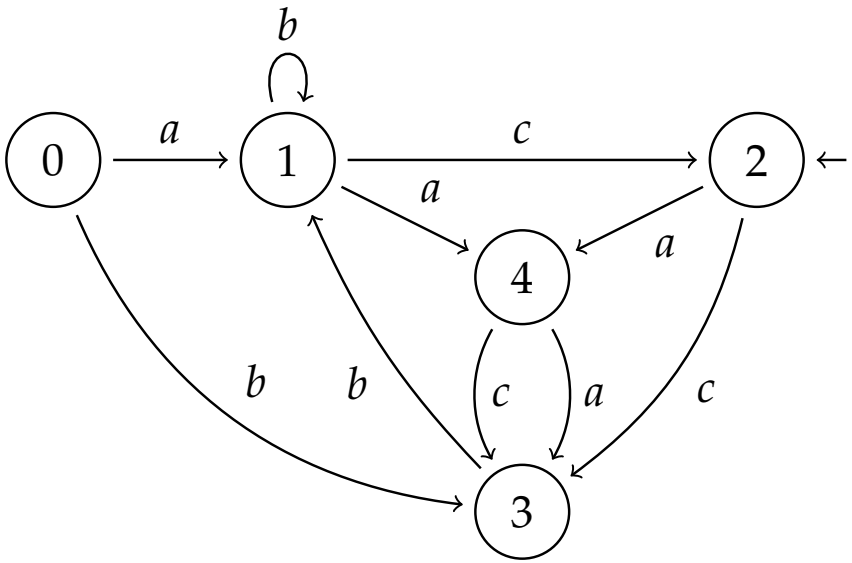
Definition (Acceptance)

An automaton *accepts* a word w if $F\ (\delta_S^*\ s\ w)$.

FCIter in action

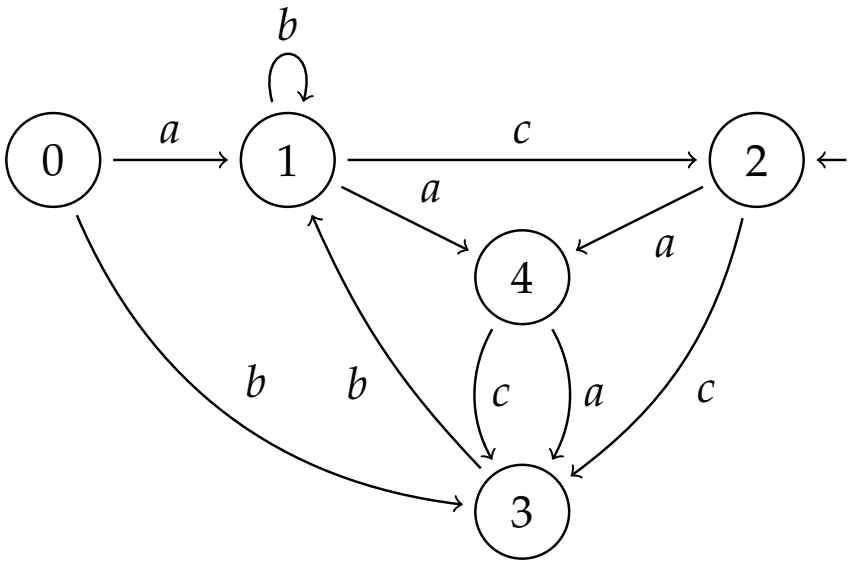


FCIter in action



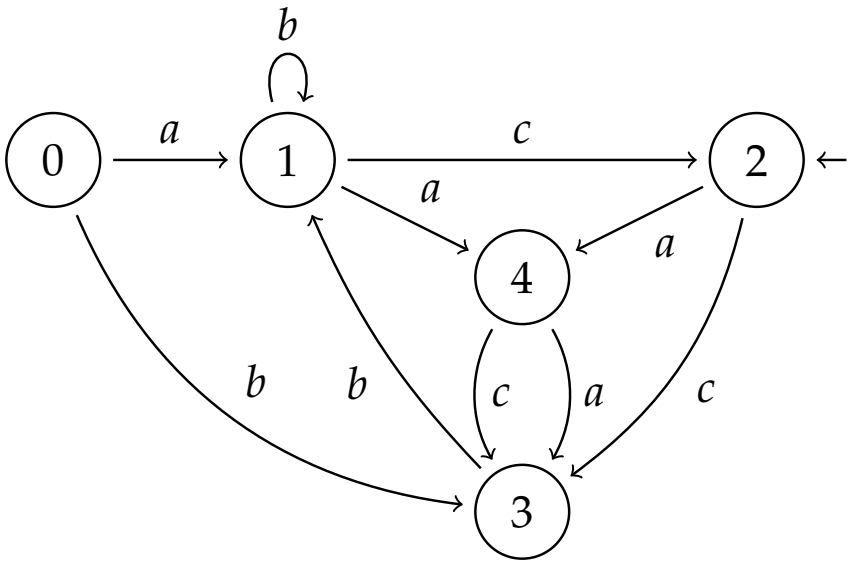
1. {2}

FCIter in action



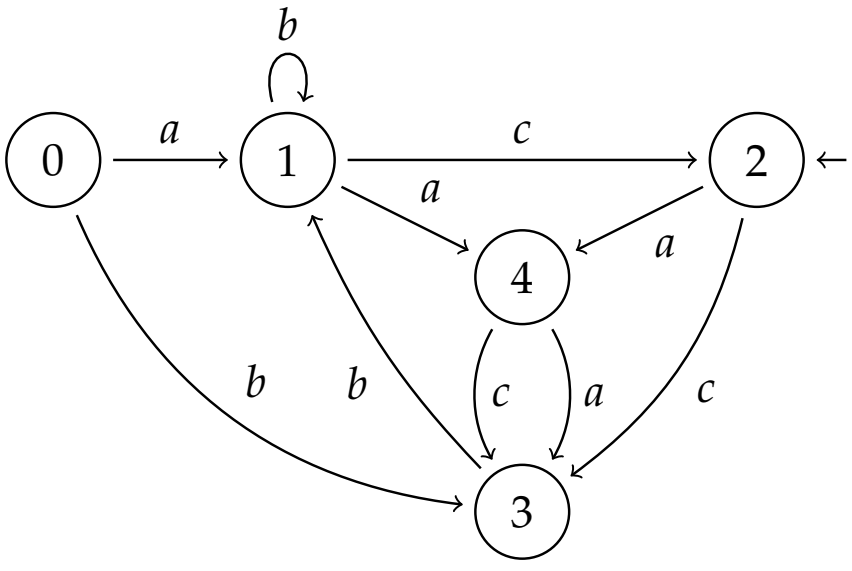
1. {2}
2. {2,3}

FCIter in action



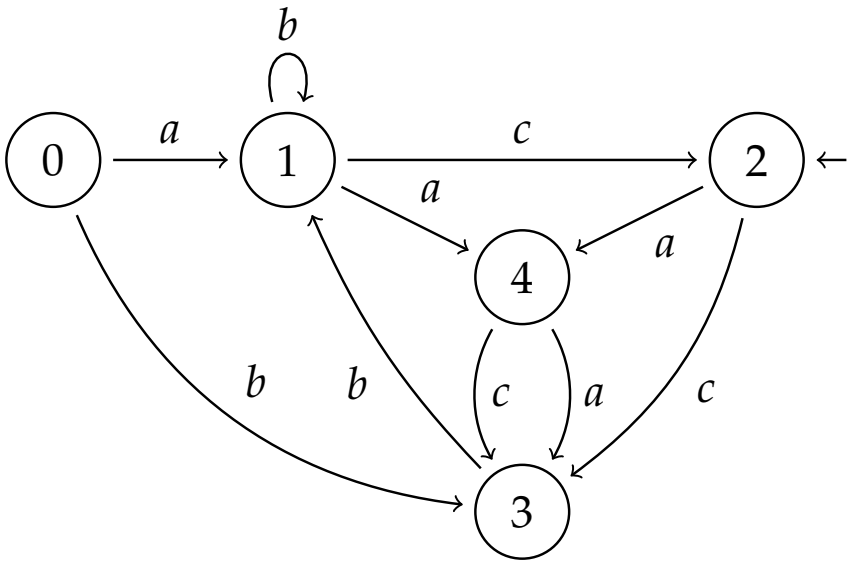
1. {2}
2. {2,3}
3. {2,3,4}

FCIter in action



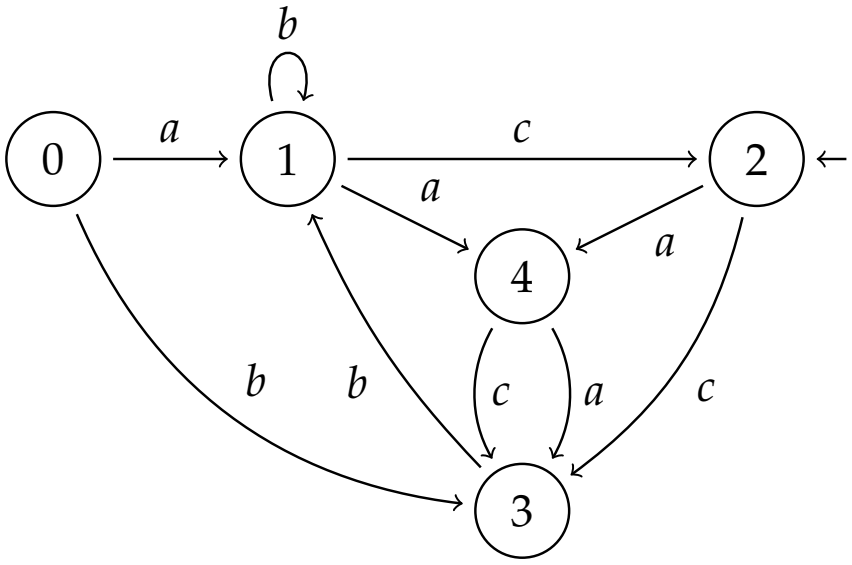
1. {2}
2. {2,3}
3. {2,3,4}
4. {2,3,4,1}

FCIter in action



1. {2}
2. {2,3}
3. {2,3,4}
4. {2,3,4,1}
5. {2,3,4,1} ← fixed point

FCIter in action

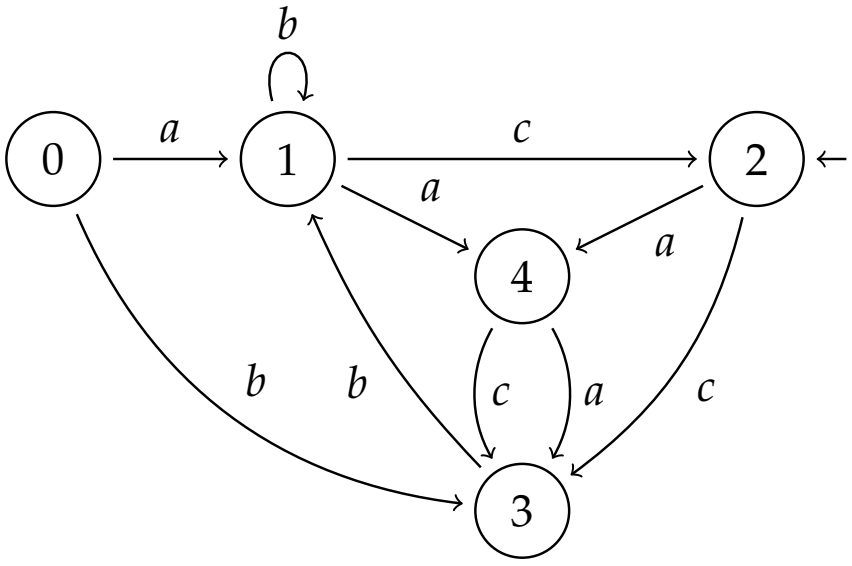


1. {2}
2. {2,3}
3. {2,3,4}
4. {2,3,4,1}
5. {2,3,4,1} ← fixed point

Allows to decide

- language is Σ^*.

FCIter in action

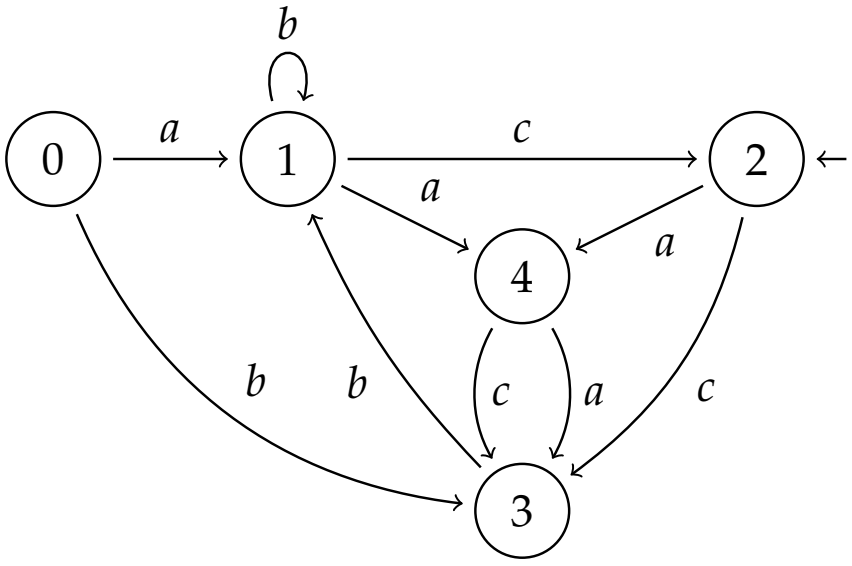


1. {2}
2. {2,3}
3. {2,3,4}
4. {2,3,4,1}
5. {2,3,4,1} ← fixed point

Allows to decide

- language is Σ^*.
- language emptiness.

FCIter in action

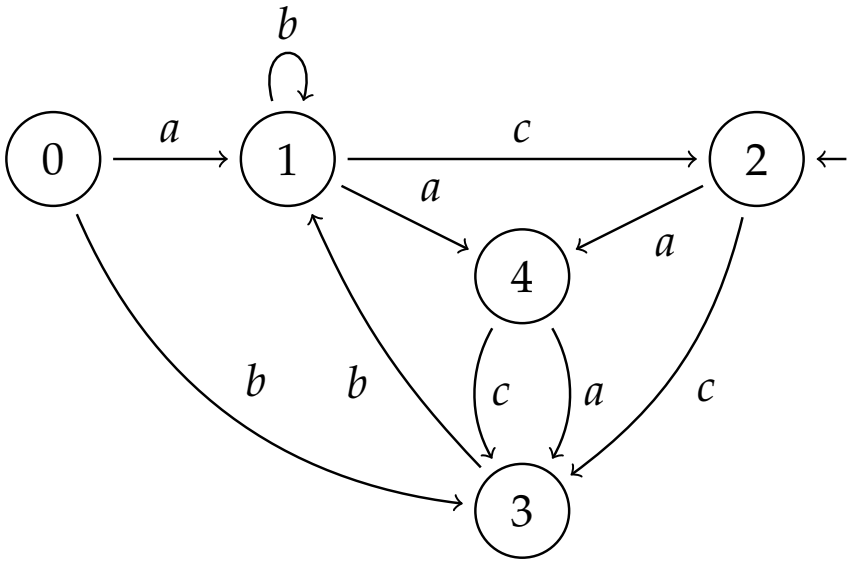


1. {2}
2. {2,3}
3. {2,3,4}
4. {2,3,4,1}
5. {2,3,4,1} ← fixed point

Allows to decide

- language is Σ^*.
- language emptiness.
- language inclusion.

FCIter in action



1. {2}
2. {2,3}
3. {2,3,4}
4. {2,3,4,1}
5. {2,3,4,1} ← fixed point

Allows to decide

- language is Σ^*.
- language emptiness.
- language inclusion.
- language equivalence.

More about automata

- Closure properties

More about automata

- Closure properties
  - Complement

More about automata

- Closure properties
  - Complement
  - Intersection

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator
- Non deterministic finite automata (NFA)

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator
- Non deterministic finite automata (NFA)
  - Equivalence of NFA and DFA

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator
- Non deterministic finite automata (NFA)
  - Equivalence of NFA and DFA
    - ★ Uses vectors

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator
- Non deterministic finite automata (NFA)
  - Equivalence of NFA and DFA
    - ★ Uses vectors
- Other constructions

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator
- Non deterministic finite automata (NFA)
  - Equivalence of NFA and DFA
    - ★ Uses vectors
- Other constructions
  - Automaton only accepting ϵ

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator
- Non deterministic finite automata (NFA)
  - Equivalence of NFA and DFA
    - ★ Uses vectors
- Other constructions
  - Automaton only accepting ϵ
  - Automaton adding some letter x to every word of a language

More about automata

- Closure properties
  - Complement
  - Intersection
  - Union
  - Difference
  - Concatenation
  - Kleene Operator
- Non deterministic finite automata (NFA)
  - Equivalence of NFA and DFA
    - ★ Uses vectors
- Other constructions
  - Automaton only accepting ϵ
  - Automaton adding some letter x to every word of a language
  - Automaton accepting some only some word w

Bibliography I

[1] Pierre Castéran and Matthieu Sozeau. *A gentle introduction to type classes and relations in Coq*. This document presents the main features of type classes and user-defined relations in the Coq proof assistant. May 2014. URL: http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf.

[2] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. "A Constructive Theory of Regular Languages in Coq". In: *Certified Programs and Proofs, Third International Conference (CPP 2013)*. Ed. by Geroges Gonthier and Michael Norrish. Vol. 8307. LNCS. Springer, Dec. 2013, pp. 82–97.

[3] Christian Doczkal and Gert Smolka. "Two-Way Automata in Coq". In: *Interative Theorem Proving (ITP 2016)*. To appear. 2016.

Bibliography II

[4] *Duality principle. Encyclopedia of Mathematics*. URL: https://www.encyclopediaofmath.org/index.php/Duality_principle.

[5] Denis Firsov and Tarmo Uustalu. "Dependently Typed Programming with Finite Sets". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. WGP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 33–44. ISBN: 978-1-4503-3810-3. DOI: 10.1145/2808098.2808102. URL: http://doi.acm.org/10.1145/2808098.2808102.

[6] François Garillot. "Generic Proof Tools and Finite Group Theory". English. Thesis. Logic in Computer Science [cs.LO]. Ecole Polytechnique X, Dec. 2011. URL: https://pastel.archives-ouvertes.fr/pastel-00649586.

Bibliography III

[7] François Garillot et al. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. Ed. by Tobias Nipkow and Christian Urban. Vol. 5674. Lecture Notes in Computer Science. Munich, Germany: Springer, 2009. URL: https://hal.inria.fr/inria-00368403.

[8] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. Inria Saclay Ile de France, 2015. URL: https://hal.inria.fr/inria-00258384.

Bibliography IV

[9] Georges Gonthier et al. "A Modular Formalisation of Finite Group Theory". In: *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs'07. Kaiserslautern, Germany: Springer-Verlag, 2007, pp. 86–101. ISBN: 3-540-74590-4, 978-3-540-74590-7. URL: http://dl.acm.org/citation.cfm?id=1792233.1792241.

[10] Michael Hedberg. "A Coherence Theorem for Martin-Löf's Type Theory". In: *J. Funct. Program.* 8.4 (July 1998), pp. 413–436. ISSN: 0956-7968. DOI: 10.1017/S095679689800315 3. URL: http://dx.doi.org/10.1017/S095679689800315 3.

[11] Dexter C. Kozen. *Automata and Computability*. 1st. Ithaca, NY, USA: Springer-Verlag New York, Inc., 1997. ISBN: 0387949070.

Bibliography V

[12] Assia Mahboubi and Enrico Tassi. "Canonical Structures for the working Coq user". In: *ITP 2013, 4th Conference on Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin, and David Pichardie. Vol. 7998. LNCS. Rennes, France: Springer, July 2013, pp. 19–34. DOI: 10.1007/978-3-642-39634-2_5. URL: https://hal.inria.fr/hal-00816703.

[13] Gert Smolka. *Base Library for ICL*. Saarland University. 2016.

[14] Gert Smolka and Chad E. Brown. "Introduction to Computational Logic. Lecture Notes SS 2014". Saarland University. 2014.

Bibliography VI

[15] Gert Smolka and Kathrin Stark. "Hereditarily Finite Sets in Constructive Type Theory". In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-27, 2016*. Ed. by Jasmin Christian Blanchette and Stephan Merz. LNCS. To appear. Springer-Verlag, 2016.

[16] Bas Spitters and Eelis van der Weegen. "Type Classes for Mathematics in Type Theory". In: *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'* 21 (2011), pp. 1–31. DOI: 10.1017/S0960129511000119. URL: http://journals.cambridge.org/action/displayAbstract?aid=8319570.

[17] Enrico Tassi and Georges Gonthier et al. *Ssreflect*. URL: http://math-comp.github.io/math-comp/.

Bibliography VII

[18] The Coq development Team. *The Coq Proof Assistant The standard library*. 2016. URL: https://coq.inria.fr/stdlib/.

The End

Thank you for your attention

Any questions? Ask away!

Extras

Definition (pure)

For a decidable predicate p with

`pure p x:= if p x then ⊤ else ⊥`

pure p is a *pure* predicate.

Reachable states in DFAs

Let A be a DFA with set of states S

We use finite closure iteration to compute reachable states.

Reachable states in DFAs

Let A be a DFA with set of states S

We use finite closure iteration to compute reachable states.

Definition (step predicate)

```
step_reach (set: list (S A)) (q : S A):=
```
$\exists\, q'\ x,\ q' \in set \to \delta_S\ q'\ x = q.$

Reachable states in DFAs

Let A be a DFA with set of states S

We use finite closure iteration to compute reachable states.

Definition (step predicate)

```
step_reach (set: list (S A)) (q : S A):=
```
$\exists\, q'\ x,\ q' \in set \to \delta_S\ q'\ x = q.$

Definition

reach `reach (q:S) := FCIter step_reach [q].`

eqTypes

```
dec (P:ℙ) := {P} + {¬ P}.

eq_dec (X:Type ) := ∀ x  y, dec (x = y).

Structure eqType := EqType {
eqtype :> Type ;
decide_eq :  eq_dec eqtype }.
```

finTypes

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

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

Admissible functions

Definition (Admissibility)

A function (f: list Y $\to$ list Y) is called *admissible* if a given list A is either a fixed-point of f or $card\ (f\ A) > card\ A$.

Admissible functions

Definition (Admissibility)

A function (f: list Y $\to$ list Y) is called *admissible* if a given list A is either a fixed-point of f or $card\ (f\ A) > card\ A$.

Theorem

Let f be an admissible function $list\ X \to list\ X$. Then

$$f^{Cardinality\ X}\ A$$

is a fixed point of f for any list A over elements of X.

FCIter

Lemma

FCStep is an admissible function.

FCIter

Lemma

FCStep is an admissible function.

FCIter

`FCIter := FCStep^Cardinality X.`

FCIter

Lemma

FCStep is an admissible function.

FCIter

`FCIter := FCStep`$^{\texttt{Cardinality X}}$.

Corollary

Let A be a list over X. Then `FCIter A` is a fixed point of *FCStep*.

A FINITE VECTOR TYPE

Goal: Construct finite type for $X_1 \longrightarrow X_2$.

A finite vector type

Goal: Construct finite type for $X_1 \longrightarrow X_2$.

Needed: List containing all vectors of type $X_1 \longrightarrow X_2$.

A finite vector type

Goal: Construct finite type for $X_1 \longrightarrow X_2$.

Needed: List containing all vectors of type $X_1 \longrightarrow X_2$.
Construct all possible images first:

A FINITE VECTOR TYPE

Goal: Construct finite type for $X_1 \longrightarrow X_2$.

Needed: List containing all vectors of type $X_1 \longrightarrow X_2$.
Construct all possible images first:

```
Fixpoint images (Y: Type ) (A: list Y) (n:  ℕ) :  list (list Y)
:=
match n with
| 0 ⇒ [[]]
| S n' ⇒ concat (map (λ x ⇒  map (cons x) (images A n')) A)
end.
```

A FINITE VECTOR TYPE

Goal: Construct finite type for $X_1 \longrightarrow X_2$.

Needed: List containing all vectors of type $X_1 \longrightarrow X_2$.

Fact

$\forall\, A, A \in \mathit{images}\ (\mathit{elem}\ X_2)\ (\mathit{Cardinality}\ X_1) \rightarrow |A| = \mathit{Cardinality}\ X_1.$

$\Rightarrow$ We can build vectors

Fact

Let A: *list* X_2 and $|A| = \mathit{Cardinality}\ X$. Then
$\mathit{count}\ (\mathit{images}\ (\mathit{elem}\ X_2)\ (\mathit{Cardinality}\ X_1))\ A = 1.$

A FINITE VECTOR TYPE

Goal: Construct finite type for $X_1 \longrightarrow X_2$.

Needed: List containing all vectors of type $X_1 \longrightarrow X_2$.

Fact

$\forall\, A, A \in \mathit{images}\ (\mathit{elem}\ X_2)\ (\mathit{Cardinality}\ X_1) \rightarrow |A| = \mathit{Cardinality}\ X_1.$

$\Rightarrow$ We can build vectors

Fact

Let A: *list* X_2 and $|A| = \mathit{Cardinality}\ X$. Then
$\mathit{count}\ (\mathit{images}\ (\mathit{elem}\ X_2)\ (\mathit{Cardinality}\ X_1))\ A = 1.$

This is enough to construct finite type $X_2^{X_1}$.