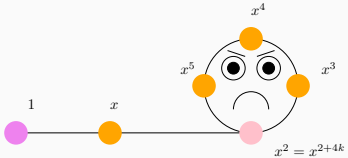


Implicit automata in typed λ-calculi I: Aperiodicity in a non-commutative logic

Nguyễn Lê Thành Dũng (Laboratoire d'informatique de Paris Nord)
and Pierre Pradic (University of Oxford)

ICALP 2020 (pre-recorded video talk)

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by
some regexp

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by
some regexp

$\varphi(w) =$ number of 'a's in w, mod 2

$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ regular $\iff$ $L = \varphi^{-1}(P)$ with

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ regular $\iff$ $L = \varphi^{-1}(P)$ with

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for Haskell, OCaml, etc.

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for Haskell, OCaml, etc.

This work in a nutshell: a functional programming characterization of star-free languages

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for Haskell, OCaml, etc.

This work in a nutshell: a functional programming characterization of star-free languages

star-free regexp

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for Haskell, OCaml, etc.

This work in a nutshell: a functional programming characterization of star-free languages

star-free regexp
also: FO[<], LTL, ...

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for Haskell, OCaml, etc.

This work in a nutshell: a functional programming characterization of star-free languages

star-free regexp
also: FO[<], LTL, ...

aperiodic finite monoids

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a, b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for Haskell, OCaml, etc.

This work in a nutshell: a functional programming characterization of star-free languages

star-free regexp
also: FO[<], LTL, ...

aperiodic finite monoids

non-commutative λ-calculus
(a.k.a. *ordered* λ-calculus)

Background: characterizations of regular languages

Regular expressions

e.g. $L = b^*(ab^*ab^*)^*$

"even number of 'a's"

Definition

$L \subseteq \Sigma^*$ regular
$\iff$
L definable by some regexp

Many other formalisms...
(e.g. NFA, MSO[<])

Finite monoids

$\varphi(w) =$ number of 'a's in w, mod 2

$$\varphi : \{a,b\}^* \to \mathbb{Z}/(2) \qquad L = \varphi^{-1}(0)$$

$$\varphi(w \cdot w') = \varphi(w) + \varphi(w')$$

Theorem (classical)

$L \subseteq \Sigma^*$ *regular* $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ morphism
- M finite monoid, $P \subseteq M$

$\approx$ deterministic finite automata

Functional programming

Implicit complexity: characterizations using high-level programming languages

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *regular* $\iff$

- L *definable by a* program $\Sigma^* \to$ *booleans* (*for a certain choice of encodings*)
- *in the* simply typed λ-calculus

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for Haskell, OCaml, etc.

This work in a nutshell: a functional programming characterization of star-free languages

star-free regexp
also: FO[<], LTL, ...

aperiodic finite monoids

non-commutative λ-calculus
(a.k.a. *ordered* λ-calculus)

Broader project: type-theoretic implicit complexity for automata theory

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\ t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\ \lambda y.\ x \qquad \texttt{false} = \lambda x.\ \lambda y.\ y$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\ t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\ \lambda y.\ x \qquad \texttt{false} = \lambda x.\ \lambda y.\ y$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\ t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\ \lambda y.\ x \qquad \texttt{false} = \lambda x.\ \lambda y.\ y$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Simple types: specifications for λ-calculus programs

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\; (f_b\; (f_b\; x))$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\ t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\ \lambda y.\ x \qquad \texttt{false} = \lambda x.\ \lambda y.\ y$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\ t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\ \lambda y.\ x \qquad \texttt{false} = \lambda x.\ \lambda y.\ y$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\; (f_b\; (f_b\; x))$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\; (f_b\; (f_b\; x))$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\; (f_b\; (f_b\; x))$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\,(f_b\,(f_b\,x))$
$\texttt{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\; (f_b\; (f_b\; x))$
$\texttt{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively
All well-typed programs terminate

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively
All well-typed programs terminate
(whereas the untyped λ-calculus is Turing-complete)

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\; (f_b\; (f_b\; x))$
$\texttt{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively
All well-typed programs terminate
(whereas the untyped λ-calculus is Turing-complete)

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\; (f_b\; (f_b\; x))$
$\texttt{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Theorem (Hillebrand & Kanellakis 1996)

The language $L \subseteq \Sigma^*$ *is* regular $\iff$
L *is defined by some* $t : \texttt{Str}_\Sigma\{o := A\} \to \texttt{Bool}$
in the simply typed λ*-calculus*
(A: *arbitrary simple type, may depend on* L)

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively
All well-typed programs terminate
(whereas the untyped λ-calculus is Turing-complete)

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\,(f_b\,(f_b\,x))$
$\texttt{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$

Theorem (Hillebrand & Kanellakis 1996)

The language $L \subseteq \Sigma^*$ *is* regular $\iff$
L *is defined by some* $t : \texttt{Str}_\Sigma\{o := A\} \to \texttt{Bool}$
in the simply typed λ*-calculus*
(A*: arbitrary simple type, may depend on* L)

($\overline{w} : \texttt{Str}_\Sigma$ therefore $\overline{w} : \texttt{Str}_\Sigma\{o := A\}$ for all A)

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively
All well-typed programs terminate
(whereas the untyped λ-calculus is Turing-complete)

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\,(f_b\,(f_b\,x))$
$\texttt{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$
$\rightsquigarrow (w \mapsto \overline{w})$ looks like a monoid morphism!

Theorem (Hillebrand & Kanellakis 1996)

The language $L \subseteq \Sigma^*$ *is* regular $\iff$
L *is defined by some* $t : \texttt{Str}_\Sigma\{o := A\} \to \texttt{Bool}$
in the simply typed λ*-calculus*
(A*: arbitrary simple type, may depend on* L)

($\overline{w} : \texttt{Str}_\Sigma$ therefore $\overline{w} : \texttt{Str}_\Sigma\{o := A\}$ for all A)

More background: Church encodings in the simply typed λ-calculus

A naive syntactic theory of functions:

$$f\,t \;\approx\; f(t) \qquad \lambda x.\, t \;\approx\; (x \mapsto t)$$

For $f : x \mapsto x^2 + 1$, we have $f(42) = 42^2 + 1$
$\rightsquigarrow$ Execution by rewriting, using substitution

$$(\lambda x.\, t)\; u \longrightarrow_\beta t\{x := u\}$$

No primitive data; encodings using functions
Church encoding of natural numbers, untyped: 1930s
Arbitrary free algebras, with types: Böhm & Berarducci 1985

Simple types: specifications for λ-calculus programs

- base type o (no constant of type o)
- $A \to B$: functions from A to B

$t : A$ ("t is of type A") defined inductively
All well-typed programs terminate
(whereas the untyped λ-calculus is Turing-complete)

Booleans

$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y$
$\texttt{Bool} = o \to (o \to o) = o \to o \to o$
so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$

(multiple arguments: $x \mapsto (y \mapsto x) \cong (x, y) \mapsto x$)

Church encoding of strings

$\overline{abb} = \lambda f_a.\, \lambda f_b.\, \lambda x.\, f_a\ (f_b\ (f_b\ x))$
$\texttt{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$

$\overline{w} \;\approx\; (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$ for $w \in \{a, b\}^*$
$\rightsquigarrow (w \mapsto \overline{w})$ looks like a monoid morphism!

Theorem (Hillebrand & Kanellakis 1996)

The language $L \subseteq \Sigma^*$ *is* regular $\iff$
L *is defined by some* $t : \texttt{Str}_\Sigma\{o := A\} \to \texttt{Bool}$
in the simply typed λ*-calculus*
(*A: arbitrary simple type, may depend on L*)

($\overline{w} : \texttt{Str}_\Sigma$ therefore $\overline{w} : \texttt{Str}_\Sigma\{o := A\}$ for all A)

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa\varnothing^c | \varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...

Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...

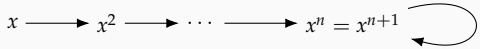
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

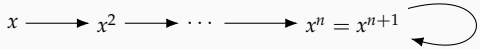
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa\varnothing^c | \varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

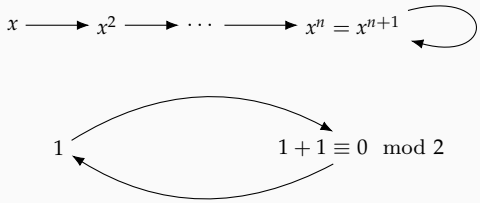
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

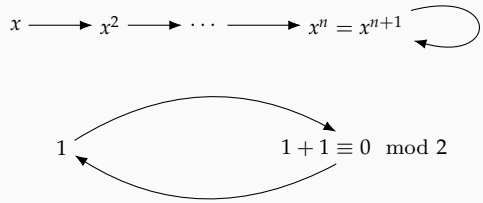
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



How to enforce aperiodicity in a typed λ-calculus?

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa \varnothing^c | \varnothing^c bb \varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

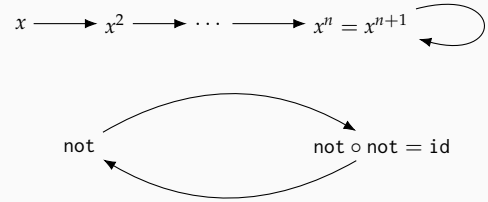
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



How to enforce aperiodicity in a typed λ-calculus?
$\texttt{not}^{n+1} \neq \texttt{not}^{n}$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa \varnothing^c | \varnothing^c bb \varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

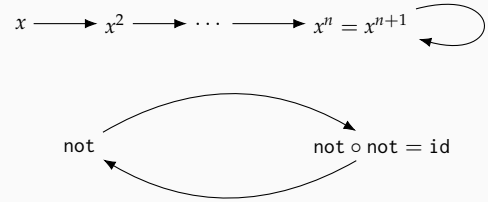
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



How to enforce aperiodicity in a typed λ-calculus?
$\texttt{not}^{n+1} \neq \texttt{not}^{n} \rightsquigarrow$ must exclude $\texttt{not} : \texttt{Bool} \to \texttt{Bool}$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa \varnothing^c | \varnothing^c bb \varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

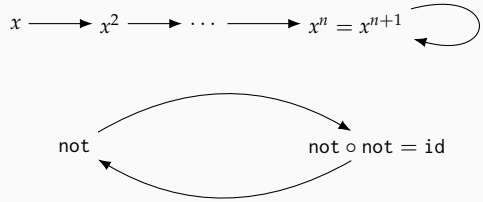
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



How to enforce aperiodicity in a typed λ-calculus?
$\texttt{not}^{n+1} \neq \texttt{not}^n \rightsquigarrow$ must exclude $\texttt{not} : \texttt{Bool} \to \texttt{Bool}$

for $b : \texttt{Bool}$, $\quad b\ x\ y \approx \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa \varnothing^c | \varnothing^c bb \varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

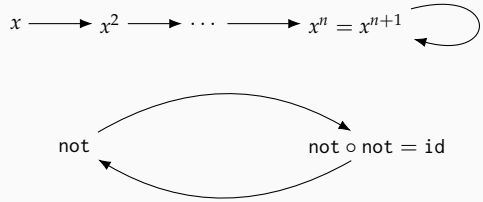
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid,* $P \subseteq M$



How to enforce aperiodicity in a typed λ-calculus?
$\texttt{not}^{n+1} \neq \texttt{not}^n \rightsquigarrow$ must exclude $\texttt{not} : \texttt{Bool} \to \texttt{Bool}$

for $b : \texttt{Bool}$, $\quad b\ x\ y \approx \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y$

$\rightsquigarrow \texttt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$

From star-free languages to non-commutativity via aperiodic monoids

Definition

Star-free language: definable by regexp

- without repetition star L^*
- with *complementation*: $L^c = \Sigma^* \setminus L$

$(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa \varnothing^c | \varnothing^c bb \varnothing^c)^c \quad (\Sigma = \{a, b\})$

However $(aa)^*$ does not have a star-free regexp...
its "minimal monoid" is $\mathbb{Z}/(2)$: *not aperiodic*

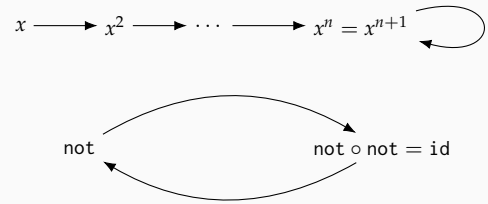
Definition

A monoid M is *aperiodic* when for any $x \in M$ there exists an exponent $n \in \mathbb{N}$ such that $x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $L = \varphi^{-1}(P)$ *with*

- $\varphi : \Sigma^* \to M$ *monoid morphism*
- M *finite and* aperiodic *monoid*, $P \subseteq M$



How to enforce aperiodicity in a typed λ-calculus?
$\texttt{not}^{n+1} \neq \texttt{not}^n \rightsquigarrow$ must exclude $\texttt{not} : \texttt{Bool} \to \texttt{Bool}$

$$\text{for } b : \texttt{Bool}, \quad b\ x\ y \ \approx\ \texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y$$

$$\rightsquigarrow \texttt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

Idea: make the order of arguments matter

"a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should not use x after y, since $\lambda x.$ occurs before λy."
$\longrightarrow$ *non-commutative* λ*-calculus*

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\rightarrow} x.\ t : A \to B$ unrestricted

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\to} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\to} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

A function $\lambda^{\circ} x.\ \lambda^{\to} y.\ \lambda^{\circ} z.\ (\ldots)$

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\to} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

A function $\lambda^{\circ} x.\ \lambda^{\to} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\rightarrow} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

A function $\lambda^{\circ} x.\ \lambda^{\rightarrow} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\to} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

A function $\lambda^{\circ} x.\ \lambda^{\to} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z
- no restrictions on y

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\rightarrow} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

$\longrightarrow$ [Polakow & Pfenning 1999] (+ weakening rule)

A function $\lambda^{\circ} x.\ \lambda^{\rightarrow} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z
- no restrictions on y

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\rightarrow} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

$\longrightarrow$ [Polakow & Pfenning 1999] (+ weakening rule)

A function $\lambda^{\circ} x.\ \lambda^{\rightarrow} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z
- no restrictions on y

Church encoding with affine types [Girard 1987]

$$\mathtt{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$$

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\to} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

$\longrightarrow$ [Polakow & Pfenning 1999] (+ weakening rule)

A function $\lambda^{\circ} x.\ \lambda^{\to} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z
- no restrictions on y

Church encoding with affine types [Girard 1987]

$\mathtt{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$

$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x))$

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\to} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

$\longrightarrow$ [Polakow & Pfenning 1999] (+ weakening rule)

A function $\lambda^{\circ} x.\ \lambda^{\to} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z
- no restrictions on y

Church encoding with affine types [Girard 1987]

$$\mathtt{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$$

$$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x))$$

Also, affine booleans: $\mathtt{Bool} = o \multimap o \multimap o$

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\to} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

$\longrightarrow$ [Polakow & Pfenning 1999] (+ weakening rule)

A function $\lambda^{\circ} x.\ \lambda^{\to} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z
- no restrictions on y

Church encoding with affine types [Girard 1987]

$$\mathtt{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$$

$$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x))$$

Also, affine booleans: $\mathtt{Bool} = o \multimap o \multimap o$

Our main theorem

The language $L \subseteq \Sigma^*$ is *star-free*
$\iff$
L is defined by some $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ-calculus

Designing a non-commutative affine λ-calculus

Idea 1: make the order of arguments matter

"$\lambda x.\ \lambda y.\ (\ldots)$ should not use x after y"

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y)$ looks OK...

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

oops, now there's a y occuring before an x...
Caused by 2 *occurrences* of z: *duplication* of $(x\ y)$

Idea 2: prohibit duplication, as in linear logic

"a function should use its arguments *at most once*"
$\longrightarrow$ non-commutative *affine* λ-calculus

But $\overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x))$ uses f_b twice!

Idea 3: let affine & unrestricted functions coexist

- $\lambda^{\rightarrow} x.\ t : A \to B$ unrestricted
- $\lambda^{\circ} x.\ t : A \multimap B$ affine

$\longrightarrow$ [Polakow & Pfenning 1999] (+ weakening rule)

A function $\lambda^{\circ} x.\ \lambda^{\rightarrow} y.\ \lambda^{\circ} z.\ (\ldots)$

- can use each of x and z at most once
- cannot use x after z
- no restrictions on y

Church encoding with affine types [Girard 1987]

$$\mathtt{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$$

$$\overline{abb} = \lambda^{\rightarrow} f_a.\ \lambda^{\rightarrow} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x))$$

Also, affine booleans: $\mathtt{Bool} = o \multimap o \multimap o$

Our main theorem

The language $L \subseteq \Sigma^*$ is *star-free*
$\iff$
L is defined by some $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ-calculus
with A *purely affine*, i.e. containing no '$\to$'

The main theorem

Theorem

The language $L \subseteq \Sigma^*$ *is* star-free

$$\Updownarrow$$

L *is defined by some* $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

The main theorem

Theorem

The language $L \subseteq \Sigma^*$ *is* star-free

$$\Updownarrow$$

L *is defined by some* $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

($\Uparrow$) Key lemma: for purely affine A, the monoid $\{t \mid t : A \multimap A\}/{=_\beta}$ is finite & aperiodic

The main theorem

Theorem

The language $L \subseteq \Sigma^*$ *is* star-free

$$\Updownarrow$$

L *is defined by some* $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

($\Uparrow$) Key lemma: for purely affine A, the monoid
$\{t \mid t : A \multimap A\}/=_\beta$ is finite & aperiodic
(in particular $\nexists\, \mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$)

The main theorem

Theorem

The language $L \subseteq \Sigma^*$ *is* star-free

$$\Updownarrow$$

L *is defined by some* $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

($\Uparrow$) Key lemma: for purely affine A, the monoid
$\{t \mid t : A \multimap A\}/=_\beta$ is finite & aperiodic
(in particular $\nexists\, \mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$)

($\Downarrow$) encoding of *aperiodic sequential transducers*
at type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$
with purely affine A

The main theorem

Theorem

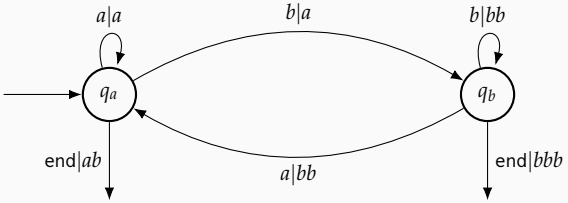
The language $L \subseteq \Sigma^*$ *is* star-free

$\Updownarrow$

L is defined by some $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

($\Uparrow$) Key lemma: for purely affine A, the monoid
$\{t \mid t : A \multimap A\}/=_\beta$ is finite & aperiodic
(in particular $\nexists\, \mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$)

($\Downarrow$) encoding of *aperiodic sequential transducers*
at type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$
with purely affine A



The main theorem

Theorem

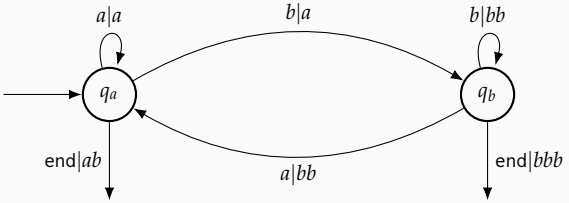
The language $L \subseteq \Sigma^*$ *is* star-free

$\Updownarrow$

L is defined by some $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

($\Uparrow$) Key lemma: for purely affine A, the monoid
$\{t \mid t : A \multimap A\}/=_\beta$ is finite & aperiodic
(in particular $\nexists \mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$)

($\Downarrow$) encoding of *aperiodic sequential transducers*
at type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$
with purely affine A
via "prime decomposition"
[Krohn & Rhodes 1965]
"prime" transducers have 2 states



The main theorem

Theorem

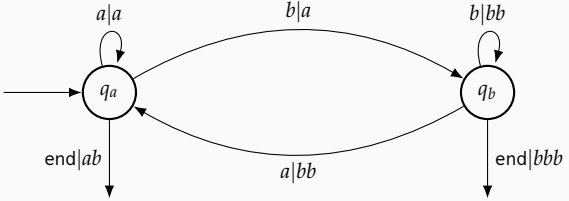
The language $L \subseteq \Sigma^*$ *is* star-free

$$\Updownarrow$$

L *is defined by some* $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

($\Uparrow$) Key lemma: for purely affine A, the monoid
$\{t \mid t : A \multimap A\}/=_\beta$ is finite & aperiodic
(in particular $\nexists\, \mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$)

($\Downarrow$) encoding of *aperiodic sequential transducers*
at type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$
with purely affine A
via "prime decomposition"
[Krohn & Rhodes 1965]
"prime" transducers have 2 states



Upcoming paper:
$\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma \iff$ aperiodic *regular* functions
(a.k.a. first-order transductions)

The main theorem

Theorem

The language $L \subseteq \Sigma^*$ *is* star-free

$\Updownarrow$

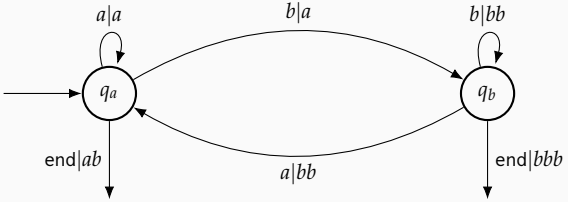
L is defined by some $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

($\Uparrow$) Key lemma: for purely affine A, the monoid
$\{t \mid t : A \multimap A\}/{=_\beta}$ is finite & aperiodic
(in particular $\nexists\, \mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$)

($\Downarrow$) encoding of *aperiodic sequential transducers*
at type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$
with purely affine A
via "prime decomposition"
[Krohn & Rhodes 1965]
"prime" transducers have 2 states

Theorem

Commutative affine types $\implies$ *regular instead of star-free*



Upcoming paper:
$\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma \iff$ aperiodic *regular* functions
(a.k.a. first-order transductions)

The main theorem

Theorem

The language $L \subseteq \Sigma^*$ *is* star-free

$$\Updownarrow$$

L *is defined by some* $t : \mathtt{Str}_\Sigma\{o := A\} \multimap \mathtt{Bool}$
in our non-commutative affine λ*-calculus*
with A purely affine, *i.e. containing no* '$\to$'

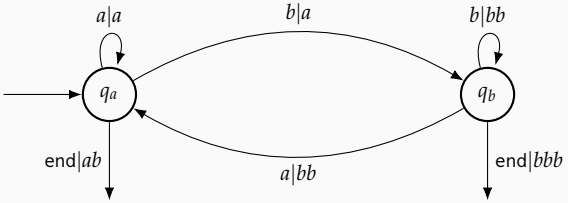
($\Uparrow$) Key lemma: for purely affine A, the monoid
$\{t \mid t : A \multimap A\}/=_\beta$ is finite & aperiodic
(in particular $\nexists\, \mathtt{not} : \mathtt{Bool} \multimap \mathtt{Bool}$)

($\Downarrow$) encoding of *aperiodic sequential transducers*
at type $\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma$
with purely affine A
via "prime decomposition"
[Krohn & Rhodes 1965]
"prime" transducers have 2 states

Theorem

Commutative affine types $\implies$ *regular instead of star-free*

$\longrightarrow$ non-commutative types are computationally meaningful



Upcoming paper:
$\mathtt{Str}_\Gamma\{o := A\} \multimap \mathtt{Str}_\Sigma \iff$ aperiodic *regular* functions
(a.k.a. first-order transductions)

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

- sequential transducers [DeYoung & Pfenning 2016]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s 2010s: *planar combinatorial maps* [Zeilberger et al.]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Further work

Transducers: functions instead of languages

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s 2010s: *planar combinatorial maps* [Zeilberger et al.]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s 2010s: *planar combinatorial maps* [Zeilberger et al.]

Further work

Transducers: functions instead of languages

- affine types $\rightsquigarrow$ regular functions

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s 2010s: *planar combinatorial maps* [Zeilberger et al.]

Further work

Transducers: functions instead of languages

- affine types $\rightsquigarrow$ regular functions *copyless* streaming string transducers

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s 2010s: *planar combinatorial maps* [Zeilberger et al.]

Further work

Transducers: functions instead of languages

- affine types $\rightsquigarrow$ regular functions
  copyless streaming string transducers
- non-commutativity $\rightsquigarrow$ FO transductions

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s 2010s: *planar combinatorial maps* [Zeilberger et al.]

Further work

Transducers: functions instead of languages

- affine types $\rightsquigarrow$ regular functions
  copyless streaming string transducers
- non-commutativity $\rightsquigarrow$ FO transductions
- tree transducers vs additive connectives

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity

  star-free $\subsetneq$ uniform AC^0

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space) [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s 2010s: *planar combinatorial maps* [Zeilberger et al.]

Further work

Transducers: functions instead of languages

- affine types $\rightsquigarrow$ regular functions *copyless* streaming string transducers
- non-commutativity $\rightsquigarrow$ FO transductions
- tree transducers vs additive connectives
- denotational semantics (Dialectica, GoI)

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity
  star-free $\subsetneq$ uniform AC^0

`https://hal.archives-ouvertes.fr/hal-02476219`

Further work

Transducers: functions instead of languages

- affine types $\rightsquigarrow$ regular functions
  copyless streaming string transducers
- non-commutativity $\rightsquigarrow$ FO transductions
- tree transducers vs additive connectives
- denotational semantics (Dialectica, GoI)

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s
  2010s: *planar combinatorial maps* [Zeilberger et al.]

Conclusion

Characterization of star-free languages using a non-commutative λ-calculus

- non-commutative types are computationally meaningful
- smallest(?) class ever captured by type-theoretic implicit complexity
  star-free $\subsetneq$ uniform AC^0

`https://hal.archives-ouvertes.fr/hal-02476219`
sequel: `https://arxiv.org/abs/1907.00467`

Further work

Transducers: functions instead of languages

- affine types $\rightsquigarrow$ regular functions
  copyless streaming string transducers
- non-commutativity $\rightsquigarrow$ FO transductions
- tree transducers vs additive connectives
- denotational semantics (Dialectica, GoI)

Previous work in "implicit automata theory"

Simply typed λ-calculus [Hillebrand & Kanellakis 1996]

Circular proofs (proofs-as-programs correspondence):

- sequential transducers [DeYoung & Pfenning 2016]
- regular languages (& logarithmic space)
  [Kuperberg, Pinault & Pous 2019, independently]

also: [Bojańczyk 2018], [Seiller 2018]

Related topics

Automata vs λ-calculus / proof theory:

- Church encodings in *higher-order model checking*
- circular proofs for Kleene algebra, μ-calculus...

Non-commutative linear logic:

- precursor: Lambek 1958 (linguistic motivations)
- *topology* of proofs/programs starting from 1980s
  2010s: *planar combinatorial maps* [Zeilberger et al.]