Implicit automata in typed λ-calculi

Lê Thành Dũng (Tito) Nguyễn — nltd@nguyentito.eu
joint work with Cécilia Pradic

Updated version of a talk given in 2021–2022

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete
$\longrightarrow$ typical "power" question: *what* does it compute?

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete $\longrightarrow$ typical "power" question: *what* does it compute? Some results known, e.g.

Theorem (Schwichtenberg 1975)

The functions $\mathbb{N}^k \to \mathbb{N}$ definable by simply-typed λ-terms $t : \mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ are the extended polynomials (generated by 0, 1, $+$, $\times$, `id` and `ifzero`).

where Nat is the type of *Church numerals*.

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete $\longrightarrow$ typical "power" question: *what* does it compute? Some results known, e.g.

Theorem (Schwichtenberg 1975)

The functions $\mathbb{N}^k \to \mathbb{N}$ definable by simply-typed λ-terms $t : \mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ are the extended polynomials (generated by 0, 1, $+$, $\times$, `id` and `ifzero`).

where Nat is the type of *Church numerals*.

But some open questions have no known satisfying answer...

Simply typed functions on Church numerals (1)

Recall that the type of Church numerals is $\mathsf{Nat} = (o \to o) \to o \to o$

$$n \in \mathbb{N} \quad \rightsquigarrow \quad \overline{n} = \lambda f.\, \lambda x.\, f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat} \text{ with } n \text{ times } f$$

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975: $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = extended polynomials

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.
What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

Simply typed functions on Church numerals (1)

Recall that the type of Church numerals is $\mathsf{Nat} = (o \to o) \to o \to o$

$$n \in \mathbb{N} \quad \rightsquigarrow \quad \overline{n} = \lambda f.\, \lambda x.\, f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat} \text{ with } n \text{ times } f$$

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$ therefore
  $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975: $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = extended polynomials

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.
What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

Simply typed functions on Church numerals (1)

Recall that the type of Church numerals is $\mathsf{Nat} = (o \to o) \to o \to o$

$$n \in \mathbb{N} \quad \rightsquigarrow \quad \overline{n} = \lambda f.\, \lambda x.\, f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat} \text{ with } n \text{ times } f$$

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$ therefore
  $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975: $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = extended polynomials

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.

What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

Simply typed functions on Church numerals (1)

Recall that the type of Church numerals is $\mathsf{Nat} = (o \to o) \to o \to o$

$$n \in \mathbb{N} \quad \rightsquigarrow \quad \overline{n} = \lambda f.\, \lambda x.\, f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat} \text{ with } n \text{ times } f$$

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$ therefore
  $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975: $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = extended polynomials

Open question

Choose some simple type A and some term $t : \mathsf{Nat}[A] \to \mathsf{Nat}$.
What functions $\mathbb{N} \to \mathbb{N}$ can be defined this way?

For $\mathsf{Nat}[A] \to \mathsf{Bool}$, there is a nice characterisation!

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

$$\overline{\texttt{011}} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{0,1\}} = (o \to o) \to (o \to o) \to o \to o$$

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

$$\overline{\mathtt{011}} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{0,1\}} = (o \to o) \to (o \to o) \to o \to o$$

Simply typed λ-terms $t : \mathsf{Str}_{\{0,1\}}[A] \to \mathsf{Bool}$ define **languages** $L \subseteq \{\mathtt{0}, \mathtt{1}\}^*$

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

$$\overline{011} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{0,1\}} = (o \to o) \to (o \to o) \to o \to o$$

Simply typed λ-terms $t : \mathsf{Str}_{\{0,1\}}[A] \to \mathsf{Bool}$ define **languages** $L \subseteq \{0,1\}^*$

Example: $t = \lambda s.\ s\ \mathtt{id}\ \mathtt{not}\ \mathtt{true} : \mathsf{Str}_{\{0,1\}}[\mathsf{Bool}] \to \mathsf{Bool}$ (even number of 1s)

$$t\ \overline{011} \longrightarrow_\beta \overline{011}\ \mathtt{id}\ \mathtt{not}\ \mathtt{true} \longrightarrow_\beta \mathtt{id}\ (\mathtt{not}\ (\mathtt{not}\ \mathtt{true})) \longrightarrow_\beta \mathtt{true}$$

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

$$\overline{011} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{0,1\}} = (o \to o) \to (o \to o) \to o \to o$$

Simply typed λ-terms $t : \mathsf{Str}_{\{0,1\}}[A] \to \mathsf{Bool}$ define **languages** $L \subseteq \{0,1\}^*$

Example: $t = \lambda s.\ s\ \mathtt{id}\ \mathtt{not}\ \mathtt{true} : \mathsf{Str}_{\{0,1\}}[\mathsf{Bool}] \to \mathsf{Bool}$ (even number of 1s)

$$t\ \overline{011} \longrightarrow_\beta \overline{011}\ \mathtt{id}\ \mathtt{not}\ \mathtt{true} \longrightarrow_\beta \mathtt{id}\ (\mathtt{not}\ (\mathtt{not}\ \mathtt{true})) \longrightarrow_\beta \mathtt{true}$$

Theorem (Hillebrand & Kanellakis 1996)

All regular languages*, and only those, can be defined this way.*

Regular languages in STλC and implicit complexity

Template for theorems at the structure/power interface

The languages/functions computed by programs of type T in the programming language $\mathcal{P}$ are exactly those in the class $\mathcal{C}$.

(dichotomy taken from Abramsky, cf. e.g. the *Structure meets Power* workshop)

- Hillebrand & Kanellakis: $\mathcal{P}$ = simply typed λ-calculus, $\mathcal{C}$ = regular languages
  - Good news: unlike "extended polynomials", a central object in another field of computer science, namely *automaton theory*
  - The definition will be recalled soon

Regular languages in STλC and implicit complexity

Template for theorems at the structure/power interface

The languages/functions computed by programs of type T in the programming language $\mathcal{P}$ are exactly those in the class $\mathcal{C}$.

(dichotomy taken from Abramsky, cf. e.g. the *Structure meets Power* workshop)

- Hillebrand & Kanellakis: $\mathcal{P}$ = simply typed λ-calculus, $\mathcal{C}$ = regular languages
  - Good news: unlike "extended polynomials", a central object in another field of computer science, namely *automaton theory*
  - The definition will be recalled soon
- *Implicit computational complexity*: $\mathcal{C}$ is a complexity class e.g. P, NP, ...
  - ICC has been an active research field since the 1990s (cf. Péchoux's HDR)
  - Historical example (Girard): $\mathcal{P}$ = Light Linear Logic, $\mathcal{C} = \mathsf{P}$ (polynomial time)

Regular languages in STλC and implicit complexity

Template for theorems at the structure/power interface

The languages/functions computed by programs of type T in the programming language $\mathcal{P}$ are exactly those in the class $\mathcal{C}$.

(dichotomy taken from Abramsky, cf. e.g. the *Structure meets Power* workshop)

- Hillebrand & Kanellakis: $\mathcal{P}$ = simply typed λ-calculus, $\mathcal{C}$ = regular languages
  - Good news: unlike "extended polynomials", a central object in another field of computer science, namely *automaton theory*
  - The definition will be recalled soon
- *Implicit computational complexity*: $\mathcal{C}$ is a complexity class e.g. P, NP, ...
  - ICC has been an active research field since the 1990s (cf. Péchoux's HDR)
  - Historical example (Girard): $\mathcal{P}$ = Light Linear Logic, $\mathcal{C} = \mathsf{P}$ (polynomial time)

Our "implicit automata" research programme: $\mathcal{C}$ coming from automaton theory

Grandeur et misère de la complexité implicite

Implicit complexity has been very successful in capturing lots of different complexity classes!

Implicit complexity has been very successful in capturing lots of different complexity classes! But the programming languages involved are often ad-hoc...

> *Several systems* [...] *have been produced; my favourite being* **LLL**, *light linear logic, which* [...] *can harbour all polytime functions. Unfortunately these systems are good for nothing, they all come from bondage: artificial restrictions on the rules which achieve certain effects, but are not justified by use, not even by some natural "semantic" considerations.* — Girard, *From Foundations to Ludics*

Grandeur et misère de la complexité implicite

Implicit complexity has been very successful in capturing lots of different complexity classes! But the programming languages involved are often ad-hoc...

> *Several systems* [...] *have been produced; my favourite being* **LLL**, *light linear logic, which* [...] *can harbour all polytime functions. Unfortunately these systems are good for nothing, they all come from bondage: artificial restrictions on the rules which achieve certain effects, but are not justified by use, not even by some natural "semantic" considerations.* — Girard, *From Foundations to Ludics*

Yet *we didn't ask* for regular languages to appear in the simply typed λ-calculus!

"Implicit automata" challenge: find *natural* characterisations for other automata-theoretic classes of languages/functions using typed λ-calculi

Grandeur et misère de la complexité implicite

Implicit complexity has been very successful in capturing lots of different complexity classes! But the programming languages involved are often ad-hoc...

> *Several systems* [...] *have been produced; my favourite being* **LLL**, *light linear logic, which* [...] *can harbour all polytime functions. Unfortunately these systems are good for nothing, they all come from bondage: artificial restrictions on the rules which achieve certain effects, but are not justified by use, not even by some natural "semantic" considerations.*
>
> — Girard, *From Foundations to Ludics*

Yet *we didn't ask* for regular languages to appear in the simply typed λ-calculus!

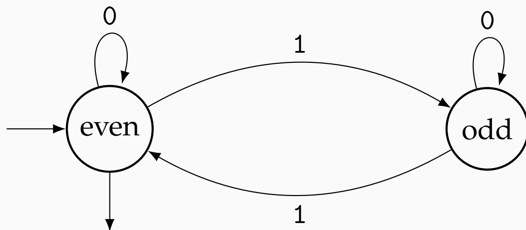
"Implicit automata" challenge: find *natural* characterisations for other automata-theoretic classes of languages/functions using typed λ-calculi

Next: we review regular languages and their generalisations to string *functions*

Regular languages

Many classical equivalent definitions (+ STλC with Church encodings!):

- *regular expressions*: `0*(10*10*)*` = "only `0`s and `1`s & even number of `1`s"
- *finite automata* (DFA/NFA): e.g. drawing below



Regular languages

Many classical equivalent definitions (+ STλC with Church encodings!):

- *regular expressions*: `0*(10*10*)*` = "only `0`s and `1`s & even number of `1`s"
- *finite automata* (DFA/NFA)
- *algebraic* definition below (very close to DFA), e.g. $M = \mathbb{Z}/(2)$

Theorem (classical)

A language $L \subseteq \Sigma^*$ is regular $\iff$ *there are a* monoid morphism $\varphi : \Sigma^* \to M$ *to a* finite monoid M *and a subset $P \subseteq M$ such that* $L = \varphi^{-1}(P) = \{w \in \Sigma^* \mid \varphi(w) \in P\}$.

Regular languages

Many classical equivalent definitions (+ STλC with Church encodings!):

- *regular expressions*: `0*(10*10*)*` = "only `0`s and `1`s & even number of `1`s"
- *finite automata* (DFA/NFA)
- *algebraic* definition below (very close to DFA), e.g. $M = \mathbb{Z}/(2)$

Theorem (classical)

A language $L \subseteq \Sigma^*$ is regular $\iff$ *there are a* monoid morphism $\varphi : \Sigma^* \to M$ *to a* finite monoid M *and a subset $P \subseteq M$ such that $L = \varphi^{-1}(P) = \{w \in \Sigma^* \mid \varphi(w) \in P\}$.*

What is the right generalization to functions $\Gamma^* \to \Sigma^*$ (*transductions*)?

Usual transduction classes (functions $\Gamma^* \to \Sigma^*$)

What is the right generalization of regular languages to *transductions*?

There are several canonical ones!

$$\underbrace{\text{sequential fonctions}}_{\textit{deterministic}\text{ finite transducers}} \subsetneq \underbrace{\text{rational functions}}_{\textit{nondeterministic}\text{ transducers}} \subsetneq \text{regular functions}$$

(“rational languages” = name for regular languages in the French tradition)

Usual transduction classes (functions $\Gamma^* \to \Sigma^*$)

What is the right generalization of regular languages to *transductions*?

There are several canonical ones!

$$\underbrace{\text{sequential fonctions}}_{\textit{deterministic}\text{ finite transducers}} \subsetneq \underbrace{\text{rational functions}}_{\textit{nondeterministic}\text{ transducers}} \subsetneq \text{regular functions}$$

("rational languages" = name for regular languages in the French tradition)

- those classes are *closed under composition*
- preservation property: L regular $\implies f^{-1}(L)$ regular

Usual transduction classes (functions $\Gamma^* \to \Sigma^*$)

What is the right generalization of regular languages to *transductions*?

There are several canonical ones!

$$\underbrace{\text{sequential fonctions}}_{\textit{deterministic}\text{ finite transducers}} \subsetneq \underbrace{\text{rational functions}}_{\textit{nondeterministic}\text{ transducers}} \subsetneq \text{regular functions}$$

("rational languages" = name for regular languages in the French tradition)

- those classes are *closed under composition*
- preservation property: L regular $\implies f^{-1}(L)$ regular

We shall be interested in *regular functions*; many definitions,

such as *streaming string transducers*

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\begin{aligned} \texttt{mapReverse} : \quad \{a,b,c,\#\}^* &\to \{a,b,c,\#\}^* \\ w_1\#\ldots\#w_n &\mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n) \end{aligned}$$

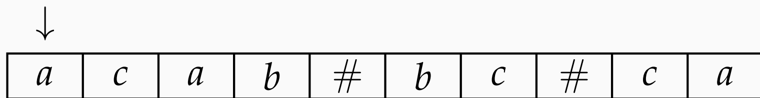
a	c	a	b	$\#$	b	c	$\#$	c	a

$X = \varepsilon \qquad Y = \varepsilon$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\begin{aligned}\texttt{mapReverse}: \quad \{a,b,c,\#\}^* &\to \{a,b,c,\#\}^* \\ w_1\#\ldots\#w_n &\mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)\end{aligned}$$

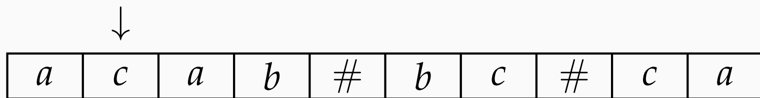


$$X = a \qquad Y = \varepsilon$$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$



$X = ca \qquad Y = \varepsilon$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\# \dots \# w_n \mapsto \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n)$$

$\downarrow$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = aca \qquad Y = \varepsilon$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\dots\#w_n \mapsto \texttt{reverse}(w_1)\#\dots\#\texttt{reverse}(w_n)$$

$\downarrow$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = baca \qquad Y = \varepsilon$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\begin{aligned} \texttt{mapReverse} : \quad \{a,b,c,\#\}^* &\to \{a,b,c,\#\}^* \\ w_1\#\ldots\#w_n &\mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n) \end{aligned}$$

$\downarrow$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = \varepsilon \qquad Y = baca\#$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\begin{array}{rrcl} \texttt{mapReverse}: & \{a,b,c,\#\}^* & \to & \{a,b,c,\#\}^* \\ & w_1\#\ldots\#w_n & \mapsto & \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n) \end{array}$$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = b \qquad Y = baca\#$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse}: \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$

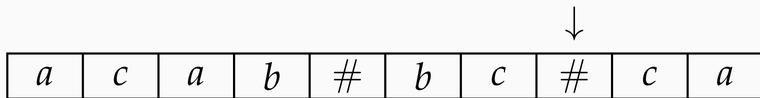
a	c	a	b	$\#$	b	c	$\#$	c	a

$X = cb \qquad Y = baca\#$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse}: \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$

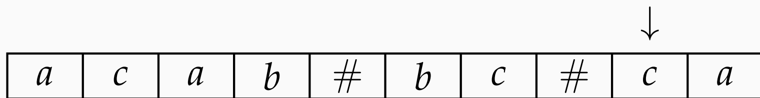


$X = \varepsilon \qquad Y = baca\#cb\#$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse} : \{a, b, c, \#\}^* \to \{a, b, c, \#\}^*$$
$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$

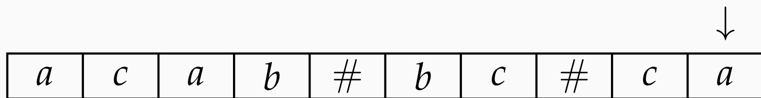


$X = c \qquad Y = baca\#cb\#$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\dots\#w_n \mapsto \texttt{reverse}(w_1)\#\dots\#\texttt{reverse}(w_n)$$



$X = ac \qquad Y = baca\#cb\#$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\# \dots \# w_n \mapsto \texttt{reverse}(w_1)\# \dots \#\texttt{reverse}(w_n)$$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = ac$ $\quad Y = baca\#cb\#$ $\quad \texttt{mapReverse}(\dots) = YX = baca\#cb\#ac$

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

$$\texttt{mapReverse} : \{a,b,c,\#\}^* \to \{a,b,c,\#\}^*$$
$$w_1\#\ldots\#w_n \mapsto \texttt{reverse}(w_1)\#\ldots\#\texttt{reverse}(w_n)$$

a	c	a	b	$\#$	b	c	$\#$	c	a

$X = ac \qquad Y = baca\#cb\# \qquad \texttt{mapReverse}(\ldots) = YX = baca\#cb\#ac$

Regular functions = computed by copyless SSTs

$$a \mapsto \begin{cases} X := aX \\ Y := Y \end{cases} \qquad \# \mapsto \begin{cases} X := \varepsilon \\ Y := YX\# \end{cases}$$

each register appears at most once on the right of a := in a transition

Linearity

Regular functions = computed by *copyless* streaming string transducers

Restrictions on "copying power": old theme in automaton theory

(par ex. Schützenberger, 1960s: rational series $\Sigma^* \to \mathbb{Z}$ with coefficients $|w|^{O(1)}$)

Linearity

Regular functions = computed by *copyless* streaming string transducers

Restrictions on “copying power”: old theme in automaton theory

(par ex. Schützenberger, 1960s: rational series $\Sigma^* \to \mathbb{Z}$ with coefficients $|w|^{O(1)}$)

λ-calculus counterpart: linear types (Girard 1987)

Here, we use the “$\lambda\ell^{\oplus\&}$-calcul” = *Dual Intuitionistic Linear Logic*
+ additive connectives $\oplus, \&$

(linear = *exactly* once; additives allow us to simulate *at most* once (*affine*))

Linearity

Regular functions = computed by *copyless* streaming string transducers

Restrictions on "copying power": old theme in automaton theory

(par ex. Schützenberger, 1960s: rational series $\Sigma^* \to \mathbb{Z}$ with coefficients $|w|^{O(1)}$)

λ-calculus counterpart: linear types (Girard 1987)

Here, we use the "$\lambda\ell^{\oplus\&}$-calcul" = *Dual Intuitionistic Linear Logic*
+ additive connectives $\oplus, \&$

(linear = *exactly* once; additives allow us to simulate *at most* once (*affine*))

$$A, B ::= o \mid \underbrace{\overbrace{A \multimap B \mid A \otimes B}^{\text{multiplicatives}}}_{\text{linear functions}} \mid \overbrace{A \mathbin{\&} B \mid A \oplus B}^{\text{additives}} \mid \underbrace{A \to B}_{\text{non-linear functions}}$$

First steps in implicit transducers

Linear Church encodings

$\mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to (o \multimap o)$, *mutatis mutandis* for Str_Σ

Definition: a type of the $\lambda\ell^{\oplus\&}$-calculus is *purely linear* if it contains no "$\to$"

Theorem

$f : \Gamma^* \to \Sigma^*$ *is* regular $\iff$ $\exists$ *a* purely linear *type A and t :* $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *in the $\lambda\ell^{\oplus\&}$-calculus such that* $\forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$

First steps in implicit transducers

Linear Church encodings

$\mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to (o \multimap o)$, *mutatis mutandis* for Str_Σ

Definition: a type of the $\lambda\ell^{\oplus\&}$-calculus is *purely linear* if it contains no "$\to$"

Theorem

$f : \Gamma^* \to \Sigma^*$ *is* regular $\iff$ $\exists$ *a* purely linear *type A and* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *in the* $\lambda\ell^{\oplus\&}$*-calculus such that* $\forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$

If we replace $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ by $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$, what class of functions do we get?

First steps in implicit transducers

Linear Church encodings

$\mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to (o \multimap o)$, *mutatis mutandis* for Str_Σ

Definition: a type of the $\lambda\ell^{\oplus\&}$-calculus is *purely linear* if it contains no "$\to$"

Theorem

$f : \Gamma^* \to \Sigma^*$ *is* regular $\iff$ $\exists$ *a* purely linear *type* A *and* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *in the* $\lambda\ell^{\oplus\&}$*-calculus such that* $\forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$

If we replace $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ by $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$, what class of functions do we get?

- Bad news: this class did not already exist in the literature

First steps in implicit transducers

Linear Church encodings

$\mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to (o \multimap o)$, *mutatis mutandis* for Str_Σ

Definition: a type of the $\lambda\ell^{\oplus\&}$-calculus is *purely linear* if it contains no "$\to$"

Theorem

$f : \Gamma^* \to \Sigma^*$ *is* regular $\iff$ $\exists$ *a* purely linear *type A and t* : $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *in the $\lambda\ell^{\oplus\&}$-calculus such that $\forall w \in \Gamma^*$, $\overline{f(w)} =_\beta t\ \overline{w}$*

If we replace $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ by $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$, what class of functions do we get?

- Bad news: this class did not already exist in the literature
- Good news: it is nonetheless natural in the context of recent work on *polyregular functions* [Bojańczyk 2018]

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*

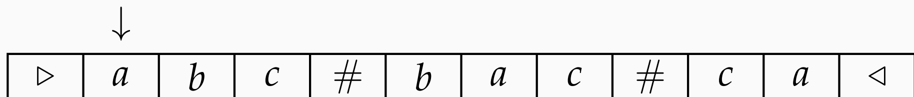
Output:

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



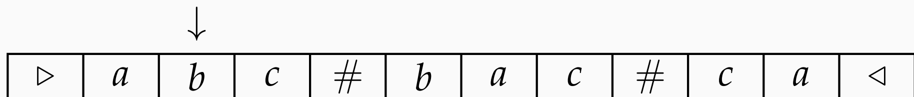
Output:

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



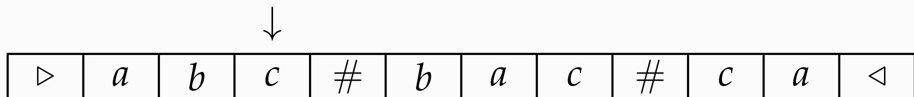
Output:

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



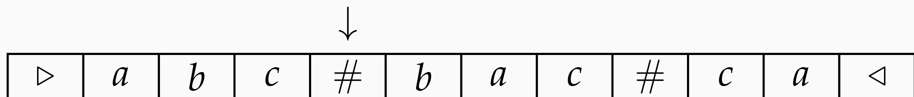
Output:

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



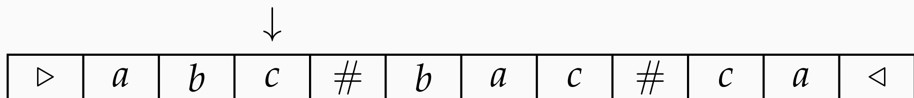
Output:

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



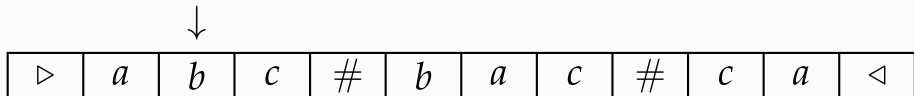
Output: c

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



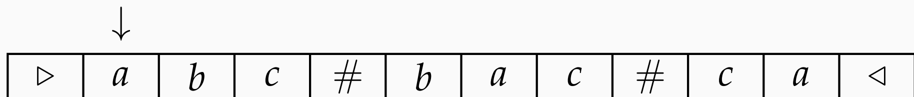
Output: cb

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



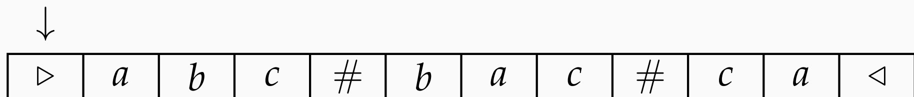
Output: cba

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



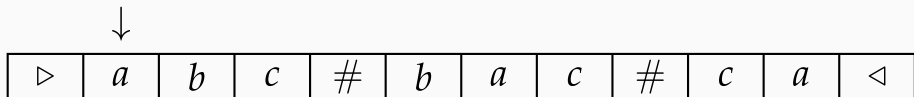
Output: *cba*

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



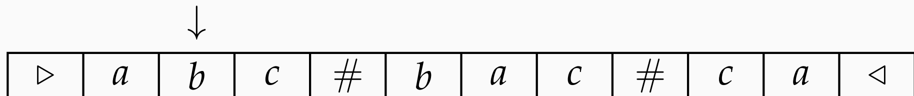
Output: *cba*

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



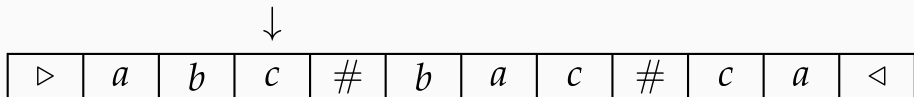
Output: *cba*

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*

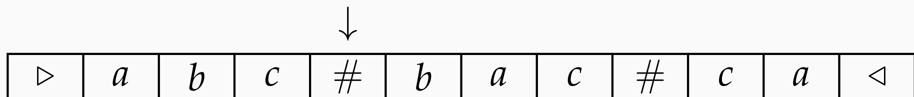


Output: *cba*

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition
$\longrightarrow$ alternative equivalent definition of *regular functions*



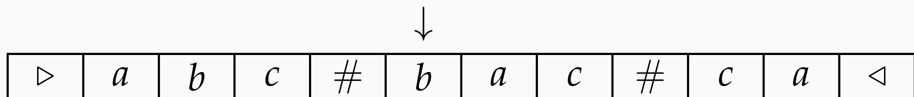
Output: $cba\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



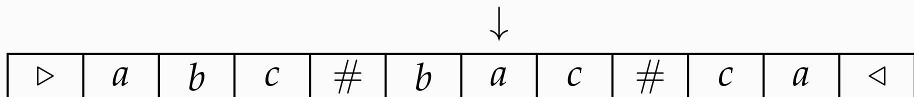
Output: $cba\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



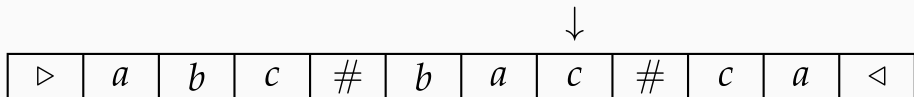
Output: $cba\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



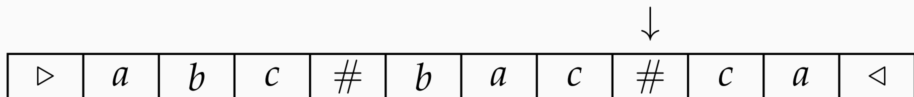
Output: $cba\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



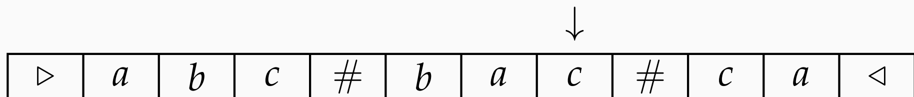
Output: $cba\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



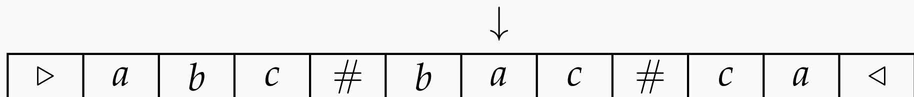
Output: $cba\#c$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



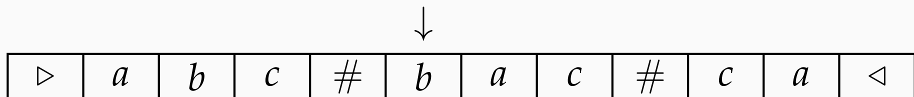
Output: $cba\#ca$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



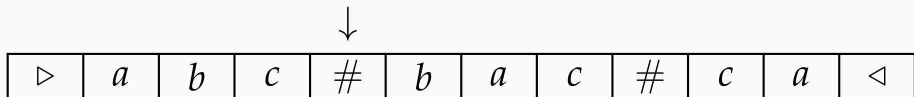
Output: $cba\#cab$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



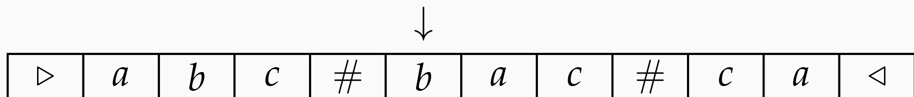
Output: $cba\#cab$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



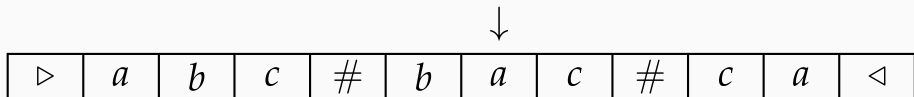
Output: $cba\#cab$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



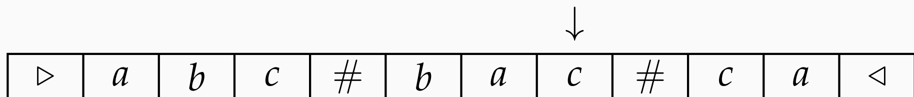
Output: *cba#cab*

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*

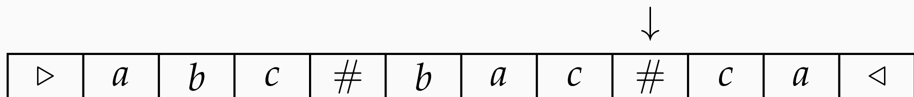


Output: $cba\#cab$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition
$\longrightarrow$ alternative equivalent definition of *regular functions*



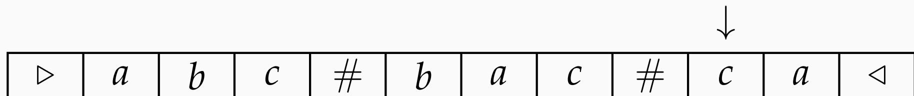
Output: $cba\#cab\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*

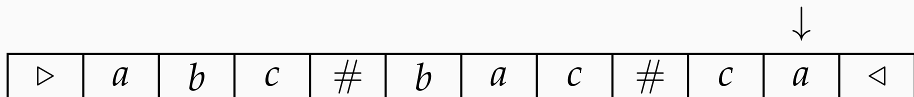


Output: $cba\#cab\#$

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



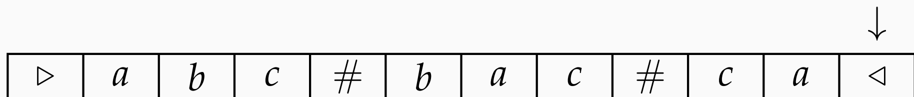
Output: $cba\#cab\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



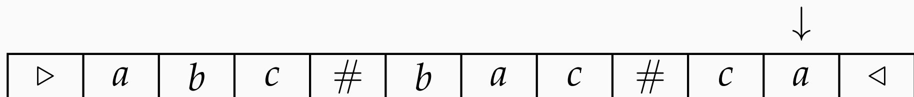
Output: $cba\#cab\#$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



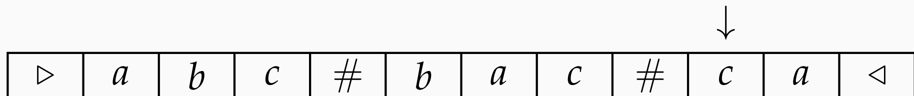
Output: $cba\#cab\#a$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



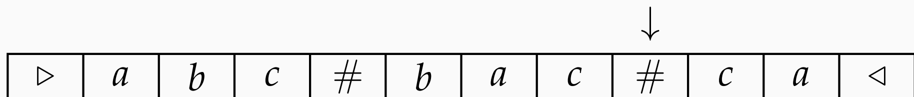
Output: *cba#cab#ac*

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



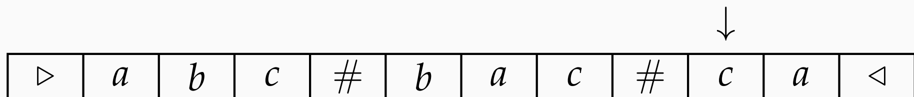
Output: $cba\#cab\#ac$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



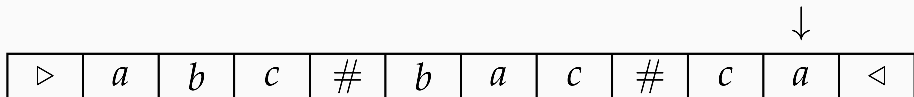
Output: $cba\#cab\#ac$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*

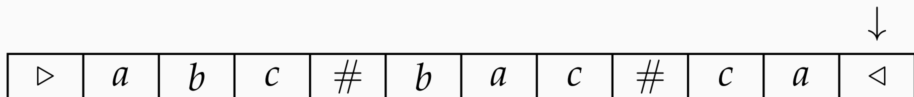


Output: $cba\#cab\#ac$

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*



Output: $cba\#cab\#ac$

A prerequisite to introduce polyregular functions...

Two-way transducers

Finite state + reading head that can move either left or right at each transition

$\longrightarrow$ alternative equivalent definition of *regular functions*

Output: $cba\#cab\#ac$

Remark: they are closely related to the *geometry of interaction* [Hines 2003], which is useful for “implicit automata” (see conclusion)

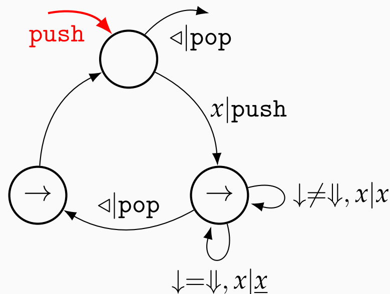
Pebble transducers [Milo, Suciu & Vianu 2000]

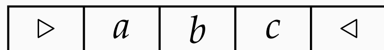
Polyregular functions = computed by <u>k-pebble transducers</u>, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:

$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





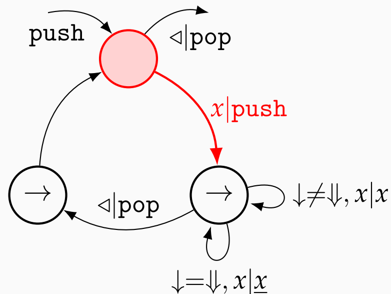
Output:

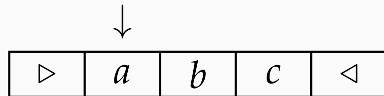
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





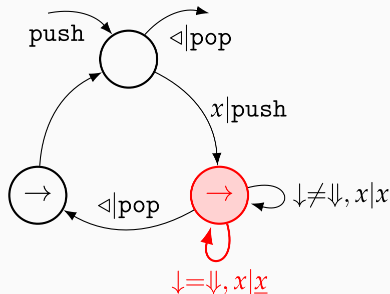
Output:

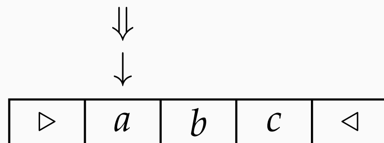
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by *k*-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





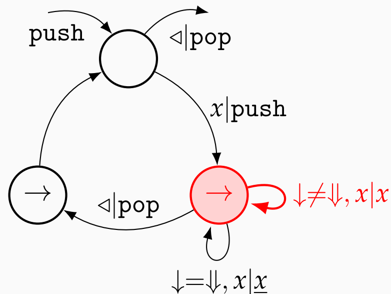
Output:

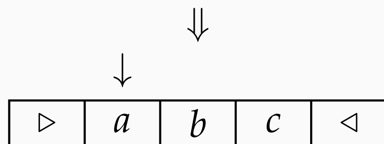
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





Output: $\underline{a}$

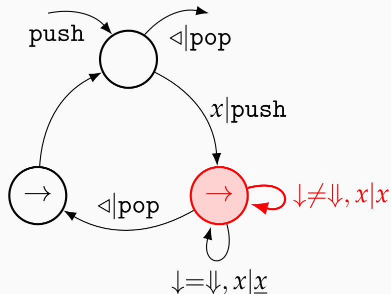
Pebble transducers [Milo, Suciu & Vianu 2000]

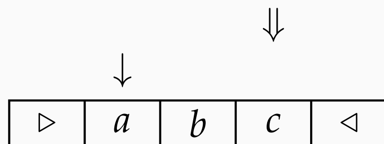
Polyregular functions = computed by <u>k-pebble transducers</u>, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:

$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





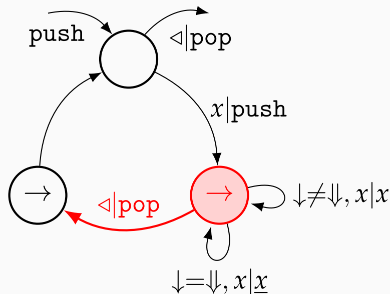
Output: $\underline{a}b$

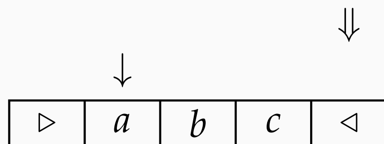
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





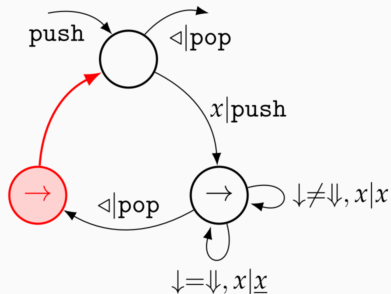
Output: $\underline{a}bc$

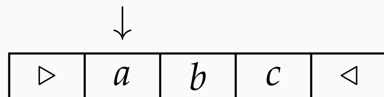
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example: $\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$





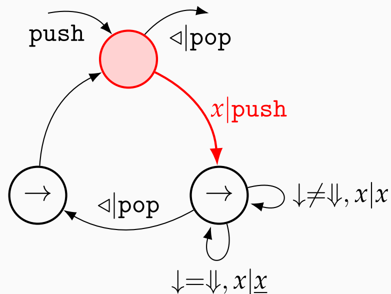
Output: $\underline{a}bc$

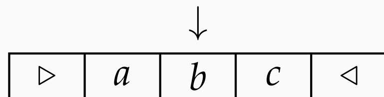
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





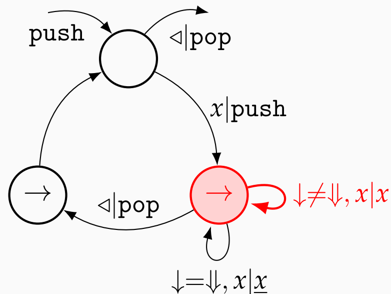
Output: $\underline{a}bc$

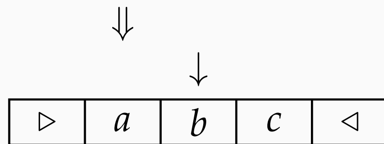
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





Output: $\underline{a}bc$

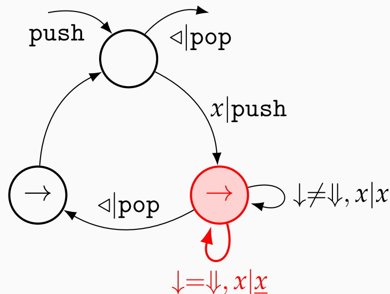
Pebble transducers [Milo, Suciu & Vianu 2000]

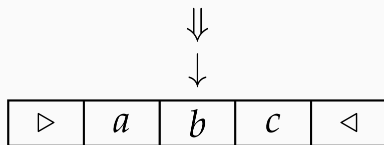
Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:

$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





Output: $\underline{a}bca$

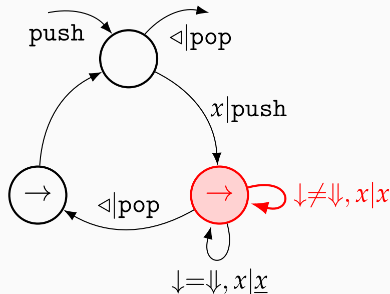
Pebble transducers [Milo, Suciu & Vianu 2000]

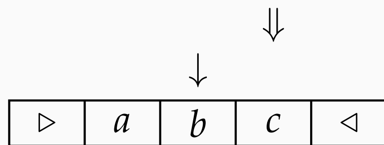
Polyregular functions = computed by <u>k-pebble transducers</u>, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:

$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





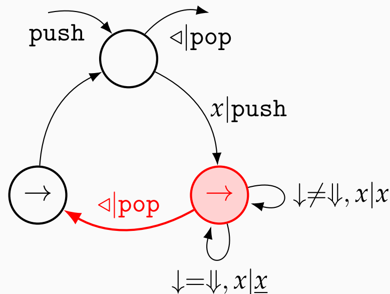
Output: $\underline{a}bca\underline{b}$

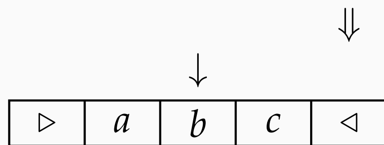
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}ca b\underline{c} \end{array}$$





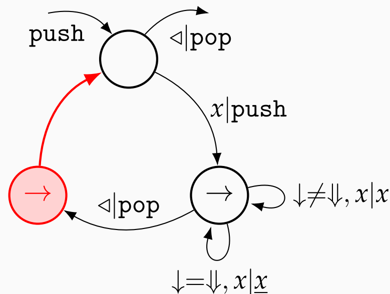
Output: $\underline{a}bca\underline{b}c$

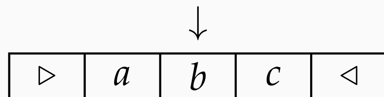
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example: $\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$





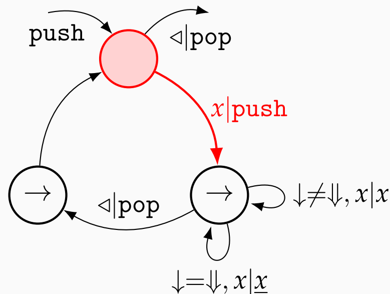
Output: $\underline{a}bca\underline{b}c$

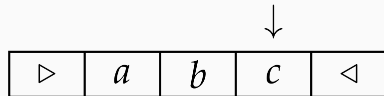
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





Output: $\underline{a}bca\underline{b}c$

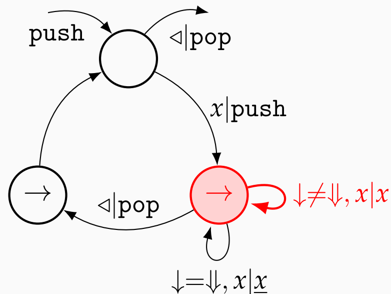
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by *k*-pebble transducers, $k \in \mathbb{N}$

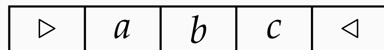
Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:

$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$



⇓



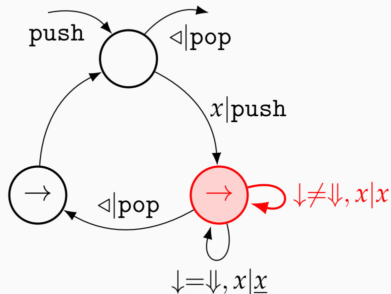
Output: $\underline{a}bca\underline{b}c$

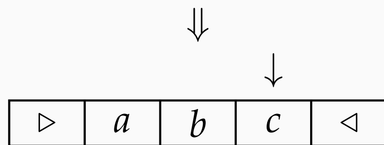
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





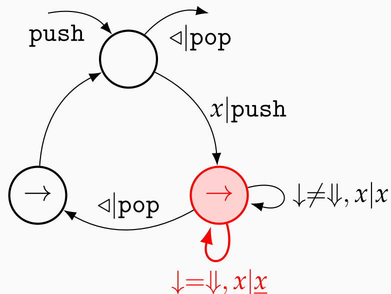
Output: $\underline{a}bca\underline{b}ca$

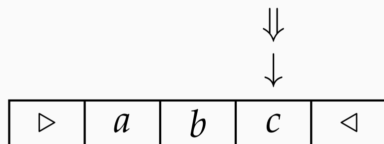
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by *k*-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





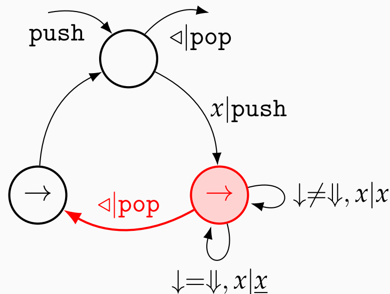
Output: $\underline{a}bca\underline{b}cab$

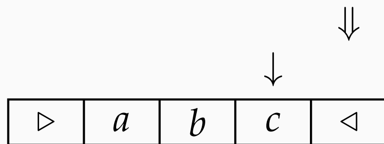
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





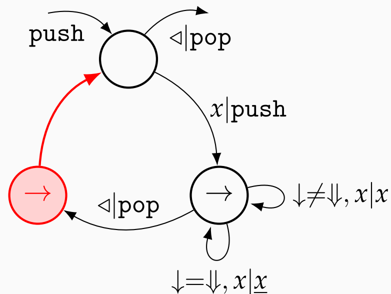
Output: $\underline{a}bca\underline{b}cab\underline{c}$

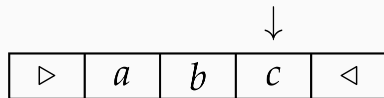
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by k-pebble transducers, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





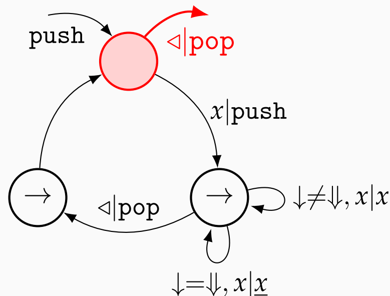
Output: $\underline{a}bca\underline{b}cab\underline{c}$

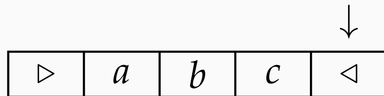
Pebble transducers [Milo, Suciu & Vianu 2000]

Polyregular functions = computed by <u>k-pebble transducers</u>, $k \in \mathbb{N}$

Finite-state control + *stack* of height $\leqslant k$ of two-way reading heads ("pebbles")

The positions of pebbles can be *compared*. Example:
$$\begin{array}{ccc} \Sigma^* & \to & (\Sigma \cup \underline{\Sigma})^* \\ abc & \mapsto & \underline{a}bca\underline{b}cab\underline{c} \end{array}$$





Output: $\underline{a}bca\underline{b}cab\underline{c}$

Comparison-free polyregular functions

Are *comparisons* between reading heads (pebbles) truly necessary?

Comparison-free polyregular functions

Are *comparisons* between reading heads (pebbles) truly necessary?

- [N., Noûs & Pradic, ICALP'21] *yes*, for instance to compute
$$f(a^n) = a\#aa\#\ldots\#a^n \text{ and } g(a^{n_1}\#\ldots\#a^{n_k}) = a^{n_1\times n_1}\#\ldots\#a^{n_k\times n_k}$$

Comparison-free polyregular functions

Are *comparisons* between reading heads (pebbles) truly necessary?

- [N., Noûs & Pradic, ICALP'21] *yes*, for instance to compute
  $$f(a^n) = a\#aa\# \dots \#a^n \text{ and } g(a^{n_1}\# \dots \#a^{n_k}) = a^{n_1 \times n_1}\# \dots \#a^{n_k \times n_k}$$
- Thus, by removing comparisons, we get a new function class.
  But is it natural/canonical?

Comparison-free polyregular functions

Are *comparisons* between reading heads (pebbles) truly necessary?

- [N., Noûs & Pradic, ICALP'21] *yes*, for instance to compute
$$f(a^n) = a\#aa\#\dots\#a^n \text{ and } g(a^{n_1}\#\dots\#a^{n_k}) = a^{n_1\times n_1}\#\dots\#a^{n_k\times n_k}$$
- Thus, by removing comparisons, we get a new function class.
  But is it natural/canonical? Yes:

Theorem

$$f : \Gamma^* \to \Sigma^* \textit{ is } \text{comparison-free polyregular}$$
$$\Longleftrightarrow$$
$$\exists A \text{ purely linear } \textit{et } t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma \textit{ tels que } \forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$$

Comparison-free polyregular functions

Are *comparisons* between reading heads (pebbles) truly necessary?

- [N., Noûs & Pradic, ICALP'21] *yes*, for instance to compute
  $$f(a^n) = a\#aa\#\dots\#a^n \text{ and } g(a^{n_1}\#\dots\#a^{n_k}) = a^{n_1\times n_1}\#\dots\#a^{n_k\times n_k}$$
- Thus, by removing comparisons, we get a new function class.
  But is it natural/canonical? Yes:

Theorem

$$f : \Gamma^* \to \Sigma^* \textit{ is } \text{comparison-free polyregular}$$
$$\iff$$
$$\exists A \text{ purely linear } \textit{et } t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma \textit{ tels que } \forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$$

ICALP'21: we study these functions from a purely automaton-theoretic POV

Are additive connectives necessary?

The additives of linear logic $\oplus/\&$ allow us:

- to represent "single use restrictions" more subtle than "copylessness" : mostly useful for *tree transducers*

  (multiplicative ($\otimes$) vs additive ($\&$) distinction implicitly appears in automata)

Are additive connectives necessary?

The additives of linear logic $\oplus/\&$ allow us:

- to represent "single use restrictions" more subtle than "copylessness" : mostly useful for *tree transducers*

  (multiplicative ($\otimes$) vs additive ($\&$) distinction implicitly appears in automata)

- to encode the *states* of an automaton! using the type $1 \oplus \cdots \oplus 1$

Are additive connectives necessary?

The additives of linear logic $\oplus/\&$ allow us:

- to represent "single use restrictions" more subtle than "copylessness" :
  mostly useful for *tree transducers*

  (multiplicative ($\otimes$) vs additive ($\&$) distinction implicitly appears in automata)
- to encode the *states* of an automaton! using the type $1 \oplus \cdots \oplus 1$

$\longrightarrow$ do we lose some computational power by removing additives?

Are additive connectives necessary?

The additives of linear logic $\oplus/\&$ allow us:

- to represent "single use restrictions" more subtle than "copylessness" : mostly useful for *tree transducers*

  (multiplicative ($\otimes$) vs additive ($\&$) distinction implicitly appears in automata)
- to encode the *states* of an automaton! using the type $1 \oplus \cdots \oplus 1$

$\longrightarrow$ do we lose some computational power by removing additives?

- on trees, *yes*, we don't even get all regular *languages* anymore (see conclusion)
- on strings, *no*, but the reason is subtle

Are additive connectives necessary?

The additives of linear logic $\oplus/\&$ allow us:

- to represent "single use restrictions" more subtle than "copylessness" : mostly useful for *tree transducers*

  (multiplicative ($\otimes$) vs additive ($\&$) distinction implicitly appears in automata)

- to encode the *states* of an automaton! using the type $1 \oplus \cdots \oplus 1$

$\longrightarrow$ do we lose some computational power by removing additives?

- on trees, *yes*, we don't even get all regular *languages* anymore (see conclusion)
- on strings, *no*, but the reason is subtle

  to compensate for the lack of states, we use *decomposition* theorems

Decomposition of regular functions

A compositional "structure" in the study of expressive "power":

Theorem (Bojańczyk et al. (cf. e.g. [Bojańczyk & Stefański 2020]))

A function is regular $\iff$ *it can be obtained as a composition of:*

- sequential *functions, see next slide*
- $\texttt{mapReverse}_\Sigma, \texttt{mapDuplicate}_\Sigma : (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ *for* $\# \notin \Sigma$

For $w_1, \ldots, w_n \in \Sigma^*$,

$$\texttt{mapReverse}_\Sigma(w_1\# \ldots \#w_n) = \texttt{rev}(w_1)\# \cdots \#\texttt{rev}(w_n)$$

$$\texttt{mapDuplicate}_\Sigma(w_1\# \ldots \#w_n) = w_1w_1\# \cdots \#w_nw_n$$

Decomposition of regular functions

A compositional “structure” in the study of expressive “power”:

Theorem (Bojańczyk et al. (cf. e.g. [Bojańczyk & Stefański 2020]))

A function is regular $\iff$ it can be obtained as a composition of:

- sequential *functions, see next slide*
- $\texttt{mapReverse}_\Sigma, \texttt{mapDuplicate}_\Sigma : (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ *for* $\# \notin \Sigma$

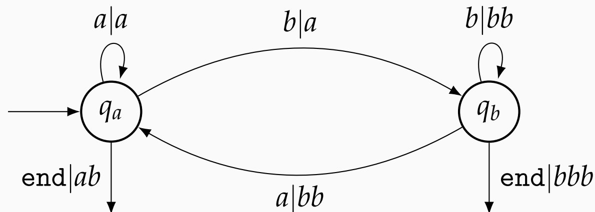
For $w_1, \ldots, w_n \in \Sigma^*$, $\texttt{mapReverse}_\Sigma(w_1\# \ldots \#w_n) = \texttt{rev}(w_1)\# \cdots \#\texttt{rev}(w_n)$

$$\texttt{mapDuplicate}_\Sigma(w_1\# \ldots \#w_n) = w_1w_1\# \cdots \#w_nw_n$$

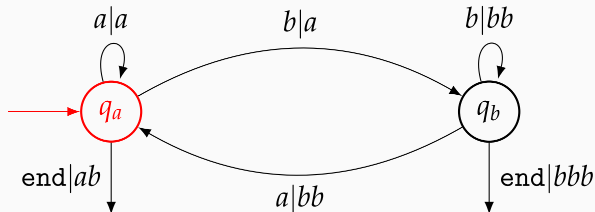
- sequential functions $\to$ correspond to finite states
- `mapReverse` and `mapDuplicate`:

  computed by single-state streaming string transducers

Sequential transducers: an example

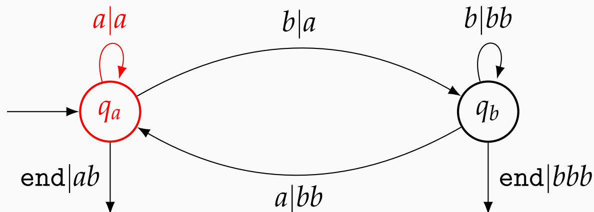


Sequential transducers: an example



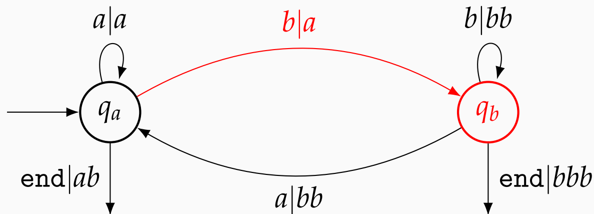
$abba \mapsto$

Sequential transducers: an example



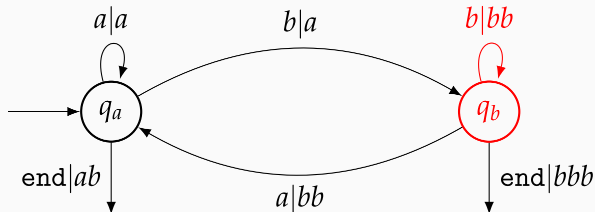
$abba \mapsto a$

Sequential transducers: an example



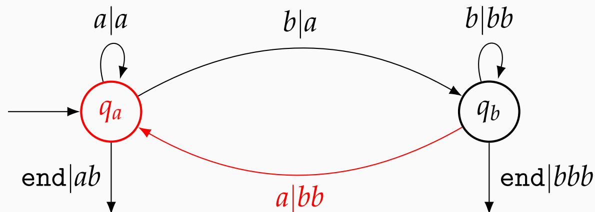
$abba \mapsto aa$

Sequential transducers: an example



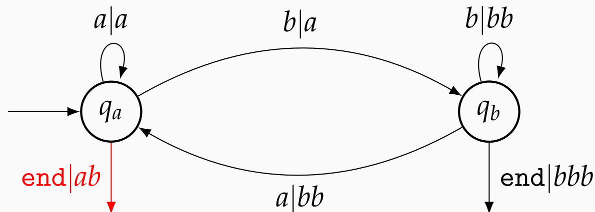
$abba \mapsto aabb$

Sequential transducers: an example



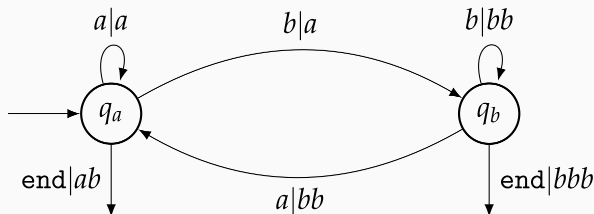
$$abba \mapsto aabbbb$$

Sequential transducers: an example



$$abba \mapsto aabbbbab$$

Sequential transducers: an example



$$abba \mapsto aabbbbab$$

Note: there is an implicit characterisation of sequential functions using *cyclic proofs* [DeYoung & Pfenning, APLAS'16]

The Krohn–Rhodes decomposition

Theorem (Krohn & Rhodes 1965, slightly reformulated)

A function is sequential $\iff$ *it can be obtained as a composition:*

- *of sequential transducers whose transitions* $Q \to Q$ *are bijective;*
- flip-flops *: seq. trans. with 2 states & w/o* $(1 \mapsto 2; 2 \mapsto 1)$ (*example in prev. slide*).

(*Fundamentally, this reflects an algebraic decomposition of finite monoids.*)

You just have to code these “basic blocks” in an *affine* λ-calculus by terms of type $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$… and then those automatically compose!

The Krohn–Rhodes decomposition

Theorem (Krohn & Rhodes 1965, slightly reformulated)

A function is sequential $\iff$ *it can be obtained as a composition:*

- *of sequential transducers whose transitions* $Q \to Q$ *are bijective;*
- flip-flops *: seq. trans. with 2 states & w/o* $(1 \mapsto 2; 2 \mapsto 1)$ (*example in prev. slide*).

(*Fundamentally, this reflects an algebraic decomposition of finite monoids.*)

You just have to code these "basic blocks" in an *affine* λ-calculus by terms of type $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$... and then those automatically compose!

Permutations on states are an important primitive

$\longrightarrow$ computational power of the *exchange rule* $f \rightsquigarrow \lambda x.\ \lambda y.\ f\ y\ x$

The Krohn–Rhodes decomposition

Theorem (Krohn & Rhodes 1965, slightly reformulated)

A function is sequential $\iff$ *it can be obtained as a composition:*

- *of sequential transducers whose transitions* $Q \to Q$ *are bijective;*
- flip-flops *: seq. trans. with 2 states & w/o* $(1 \mapsto 2; 2 \mapsto 1)$ (*example in prev. slide*).

(*Fundamentally, this reflects an algebraic decomposition of finite monoids.*)

You just have to code these "basic blocks" in an *affine* λ-calculus by terms of type $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$... and then those automatically compose!

Permutations on states are an important primitive

$\longrightarrow$ computational power of the *exchange rule* $f \rightsquigarrow \lambda x.\ \lambda y.\ f\ y\ x$

Connection between Krohn–Rhodes without bijections and *non-commutative* types?

Aperiodicity

Theorem (Krohn & Rhodes 1965, aperiodic case)

A function is aperiodic sequential $\iff$ *it is a composition of "flip-flops".*

Aperiodicity

Theorem (Krohn & Rhodes 1965, aperiodic case)

A function is aperiodic sequential $\iff$ *it is a composition of "flip-flops".*

Aperiodic sequential transducer = its transition monoid is aperiodic

Definition

A monoid M is *aperiodic* when $\forall x \in M,\ \exists n \in \mathbb{N} : x^n = x^{n+1}$.

Aperiodicity

Theorem (Krohn & Rhodes 1965, aperiodic case)

A function is aperiodic sequential $\iff$ *it is a composition of "flip-flops".*

Aperiodic sequential transducer = its transition monoid is aperiodic

Definition

A monoid M is *aperiodic* when $\forall x \in M,\ \exists n \in \mathbb{N} : x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

A language $L \subseteq \Sigma^*$ is star-free $\iff$ $\exists$ *a* morphism $\varphi : \Sigma^* \to M$ *to a* finite and aperiodic monoid M *and $P \subseteq M$ such that $L = \varphi^{-1}(P)$.*

(Reminder : w/o "aperiodic" $\rightsquigarrow$ regular languages)

Star-free languages

A monoid M is *aperiodic* when $\forall x \in M,\ \exists n \in \mathbb{N} : x^n = x^{n+1}$.

Theorem (Schützenberger 1965)

A language $L \subseteq \Sigma^*$ is star-free $\iff$ $\exists$ *a* morphism $\varphi : \Sigma^* \to M$ *to a* finite and aperiodic monoid M *and* $P \subseteq M$ *such that* $L = \varphi^{-1}(P)$.

Regular expressions w/o (Kleene) *star* – but with *complementation*:

$$L, L' ::= \varnothing \mid \{a\} \mid L \cdot L' \mid L \cup L' \mid L^{\mathrm{c}}$$

for instance $(ab)^* = (b\varnothing^c \cup \varnothing^c a \cup \varnothing^c aa\varnothing^c \cup \varnothing^c bb\varnothing^c)^c$, but $(aa)^*$ is not star-free:
its “minimal monoid” $\mathbb{Z}/2\mathbb{Z}$ is not aperiodic

Implicit automata in non-commutative logic

Star-free regexp: $L, L' ::= \varnothing \mid \{a\} \mid L \cdot L' \mid L \cup L' \mid L^c$

Theorem (N. & Pradic, ICALP 2020)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $\exists$ *a* purely affine *type* A *and* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Bool}$ *in the "$\lambda\wp$-calcul" such that* $L = \{w \in \Sigma^* \mid t\ \overline{w} =_{\beta\eta} \mathtt{true}\}$

Implicit automata in non-commutative logic

Star-free regexp: $L, L' ::= \varnothing \mid \{a\} \mid L \cdot L' \mid L \cup L' \mid L^c$

Theorem (N. & Pradic, ICALP 2020)

$L \subseteq \Sigma^*$ *is* star-free $\iff$ $\exists$ *a* purely affine *type* A *and* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Bool}$ *in the "*$\lambda\wp$*-calcul" such that* $L = \{w \in \Sigma^* \mid t\ \overline{w} =_{\beta\eta} \mathtt{true}\}$

$\lambda\wp$-calcul : affine variant of *Intuitionistic Non-Commutative Linear Logic*

(Jeff Polakow's PhD thesis, dir. Franck Pfenning)

Idea (going back to Lambek 1958) : the order of arguments matters

"a function $\lambda x.\ \lambda y.\ (\ldots)$ cannot use x *after* y"

$\longrightarrow$ typing judgments $\Phi \mid \Psi \vdash t : A$ for a *set* Φ (non-linear context)

and and *ordered list* Ψ (linear context)

How to go from typed λ-terms to automata?

Semantic evaluation and categorical automaton theory

How to go from typed λ-terms to automata?

Through *denotational semantics* : translation λ-terms $\rightsquigarrow$ algebraic structures

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

How to go from typed λ-terms to automata?

Through *denotational semantics* : translation λ-terms $\rightsquigarrow$ algebraic structures

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

Approach for our "implicit transducer" results

- Find a *monoidal closed category $\mathcal{C}$*
  (provides a semantics for the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus)

Semantic evaluation and categorical automaton theory

How to go from typed λ-terms to automata?

Through *denotational semantics* : translation λ-terms $\rightsquigarrow$ algebraic structures

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

Approach for our "implicit transducer" results

- Find a *monoidal closed category* $\mathcal{C}$
  (provides a semantics for the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus)
- such that $\mathcal{C}$*-automata* compute regular functions
  (notion of automaton over a category used here: [Colcombet & Petrişan 2017])

Semantic evaluation and categorical automaton theory

How to go from typed λ-terms to automata?

Through *denotational semantics* : translation λ-terms $\rightsquigarrow$ algebraic structures

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

Approach for our "implicit transducer" results

- Find a *monoidal closed category* $\mathcal{C}$
  (provides a semantics for the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus)
- such that $\mathcal{C}$*-automata* compute regular functions
  (notion of automaton over a category used here: [Colcombet & Petrişan 2017])

e.g. $\mathcal{C} = \mathsf{Int}(\mathsf{PFinSet}) \rightsquigarrow$ two-way automata [Hines 2003]
$\longrightarrow$ connection with geometry of interaction, but does not handle additives

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_\oplus$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_\oplus$ = free finite coproduct completion $\simeq$ adds finite states

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_\oplus$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_\oplus$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_\oplus$ is merely "partially" monoidal closed...

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_{\oplus}$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_{\oplus}$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_{\oplus}$ is merely "partially" monoidal closed...
- However, $(\mathcal{SR}_{\&})_{\oplus}$ is monoidal closed ($(-)_{\&}$ = product completion)

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_\oplus$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_\oplus$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_\oplus$ is merely "partially" monoidal closed...
- However, $(\mathcal{SR}_\&)_\oplus$ is monoidal closed ($(-)_\&$ = product completion)

$((-)_\&)_\oplus$ completion $\simeq$ adding finite states with non-determinism

- copyless SSTs can be determinised [Alur & Deshmukh 2011]

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_\oplus$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_\oplus$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_\oplus$ is merely "partially" monoidal closed...
- However, $(\mathcal{SR}_\&)_\oplus$ is monoidal closed ($(-)_\&$ = product completion)

$((-)_\&)_\oplus$ completion $\simeq$ adding finite states with non-determinism

- copyless SSTs can be determinised [Alur & Deshmukh 2011]
- **our work:** *categorical* determinisation using the existence of some "function spaces" $A \multimap B$ = "partielle" monoidal closure!

Monoidal closed structure on the category $(\mathcal{SR}_{\&})_{\oplus}$

For $A, B \in \mathsf{Obj}(\mathcal{SR})$: decompose morphisms

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \quad \rightsquigarrow \quad \text{forme} \begin{cases} X := Z_1 X Z_2 Y \\ Y := Z_3 \end{cases} \quad + \quad \text{labels } Z_1 = ab, \ldots$$

copyless $\Longrightarrow$ *finite* number of possible shapes $\Longrightarrow$ $A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$

Monoidal closed structure on the category $(\mathcal{SR}_{\&})_{\oplus}$

For $A, B \in \mathsf{Obj}(\mathcal{SR})$: decompose morphisms

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \quad \rightsquigarrow \quad \text{forme} \begin{cases} X := Z_1 X Z_2 Y \\ Y := Z_3 \end{cases} \quad + \quad \text{labels } Z_1 = ab, \ldots$$

$$\textit{copyless} \implies \textit{finite} \text{ number of possible shapes} \implies A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$$

This decomposition is a classical technique, also used to show that comparison-free polyregular functions are closed under composition in our ICALP'21 paper.

Monoidal closed structure on the category $(\mathcal{SR}_{\&})_{\oplus}$

In $(\mathcal{C}_{\&})_{\oplus}$, we have $\bigoplus_u \&_x A_{u,x} \multimap \bigoplus_v \&_y B_{v,y} \;=\; \&_u \bigoplus_v \&_y \bigoplus_x A_{u,x} \multimap B_{v,y}$

Theorem ("Dialectica-like" construction inspired by [Gödel, de Paiva, Hofstra])

Let $\mathcal{C}$ a symmetric monoidal category. If the function space $A \multimap B$ exists in $\mathcal{C}_{\oplus}$ for all $A, B \in \mathsf{Obj}(\mathcal{C})$, then $(\mathcal{C}_{\&})_{\oplus}$ is monoidal closed.

For $A, B \in \mathsf{Obj}(\mathcal{SR})$: decompose morphisms

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \quad \rightsquigarrow \quad \text{forme} \begin{cases} X := Z_1 X Z_2 Y \\ Y := Z_3 \end{cases} \quad + \quad \text{labels } Z_1 = ab, \ldots$$

$$\textit{copyless} \implies \textit{finite} \text{ number of possible shapes} \implies A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$$

This decomposition is a classical technique, also used to show that comparison-free polyregular functions are closed under composition in our ICALP'21 paper.

Conclusion

- characterisations of transduction classes in a linear λ-calculus
  + star-free languages via a non-commutative type system
- connections between denotational semantics & categorical automaton theory
- discovery of the class of comparison-free polyregular functions

Conclusion

- characterisations of transduction classes in a linear λ-calculus
  + star-free languages via a non-commutative type system
- connections between denotational semantics & categorical automaton theory
- discovery of the class of comparison-free polyregular functions

Conclusion

- characterisations of transduction classes in a linear λ-calculus

  + star-free languages via a non-commutative type system
- connections between denotational semantics & categorical automaton theory
- discovery of the class of comparison-free polyregular functions

A lot of perspectives!

- old open problem: $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ in simply typed λ-calculus

  $\longrightarrow$ connections with higher-order model checking?

Conclusion

- characterisations of transduction classes in a linear λ-calculus
  + star-free languages via a non-commutative type system
- connections between denotational semantics & categorical automaton theory
- discovery of the class of comparison-free polyregular functions

A lot of perspectives!

- old open problem: $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ in simply typed λ-calculus
  $\longrightarrow$ connections with higher-order model checking?
- affine λ-c. w/o additives $\underbrace{\subseteq}_{\text{geometry of interaction}}$ tree-walking automata $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ reg. *tree* lang.

Conclusion

- characterisations of transduction classes in a linear λ-calculus
  + star-free languages via a non-commutative type system
- connections between denotational semantics & categorical automaton theory
- discovery of the class of comparison-free polyregular functions

A lot of perspectives!

- old open problem: $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ in simply typed λ-calculus
  $\longrightarrow$ connections with higher-order model checking?
- affine λ-c. w/o additives $\underbrace{\subseteq}_{\text{geometry of interaction}}$ tree-walking automata $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ reg. *tree* lang.
- atoms (nominal sets), parsimonious λ-calculus, polymorphism, ...

Conclusion

Thanks for your attention!

- characterisations of transduction classes in a linear λ-calculus

  + star-free languages via a non-commutative type system
- connections between denotational semantics & categorical automaton theory
- discovery of the class of comparison-free polyregular functions

A lot of perspectives!

- old open problem: $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ in simply typed λ-calculus

  $\longrightarrow$ connections with higher-order model checking?
- affine λ-c. w/o additives $\underbrace{\subseteq}_{\text{geometry of interaction}}$ tree-walking automata $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ reg. *tree* lang.
- atoms (nominal sets), parsimonious λ-calculus, polymorphism, ...