Introduction to implicit automata in typed λ-calculi and higher-order transducers

Lê Thành Dũng (Tito) Nguyễn (CNRS & Aix-Marseille Univ.)
based on joint work with several people, mainly **Cécilia Pradic** (*looking for a new job!*)

Higher-Order Computation in Implicit and Descriptive Complexity
26-27 May 2026, IRIF, Paris

Two convergent lines of work about *relating the expressive power of automata-based and λ-calculus-based formalisms*

- **Implicit automata:** motivated by questions internal to λ-calculus, and by Implicit Computational Complexity (ICC)

Template for ICC theorems

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

- Active research field since the 1990s, cf. Péchoux's HDR
- Historical example (Girard): $\mathcal{P}$ = Light Linear Logic, $\mathcal{C} = \mathsf{P}$ (polynomial time)
- Basic idea here: $\mathcal{C} \in$ automata theory

Two convergent lines of work about *relating the expressive power of automata-based and λ-calculus-based formalisms*

- **Implicit automata:** motivated by questions internal to λ-calculus, and by Implicit Computational Complexity (ICC)

Template for ICC theorems

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

  - Active research field since the 1990s, cf. Péchoux's HDR
  - Historical example (Girard): $\mathcal{P}$ = Light Linear Logic, $\mathcal{C} = \mathsf{P}$ (polynomial time)
  - Basic idea here: $\mathcal{C} \in$ automata theory
- **Higher-order grammars and transducers:** applying λ-calculus to motivations from automata and formal languages
  - older tradition (Damm / Engelfriet & Vogler, 1980s)
  - 21st c. revival (Kanazawa '08; Gallot, Lemay & Salvati '20; ...): linear types

Grandeur et misère de la complexité implicite

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

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.* — J.-Y. 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.* — J.-Y. Girard, *From Foundations to Ludics*

What about the simply typed λ-calculus?

- Hillebrand, Kanellakis & Mairson 1993: ELEMENTARY
  - "matches" TOWER-completeness of normalization [Statman 1979; Schmitz 2016]
  - using ad-hoc encodings (motivation: database queries)

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.* — J.-Y. Girard, *From Foundations to Ludics*

What about the simply typed λ-calculus?

- Hillebrand, Kanellakis & Mairson 1993: ELEMENTARY
  - "matches" TOWER-completeness of normalization [Statman 1979; Schmitz 2016]
  - using ad-hoc encodings (motivation: database queries)
- What about the standard *Church encodings?*

Simply typed λ-terms operating on Church encodings

Some results known, e.g. Leivant 1993, Zaionc 1994

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*: $\mathsf{Nat} = (o \to o) \to o \to o$

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

All inhabitants of Nat are equal to some $\overline{n}$ up to $=_{\beta\eta}$

Simply typed λ-terms operating on Church encodings

Some results known, e.g. Leivant 1993, Zaionc 1994

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*: $\mathsf{Nat} = (o \to o) \to o \to o$

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

All inhabitants of Nat are equal to some $\overline{n}$ up to $=_{\beta\eta}$

Let's add a bit of (meta-level) polymorphism: $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o]$ for $n \in \mathbb{N}$

More difficult question (what is the right perspective on it?)

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?

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; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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

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; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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

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

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

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; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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

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

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

Theorem (Hillebrand & Kanellakis 1996)

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

Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A and any simply typed λ-term $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$, the language $\mathcal{L}(t) = \{w \in \Sigma^* \mid t\,\overline{w} \to_\beta^* \mathtt{true}\}$ is regular.

Part 1 of proof.

Fix type A. Any *denotational semantics* $[\![-]\!]$ quotients words:

$$w \in \Sigma^* \rightsquigarrow \overline{w} : \mathsf{Str}[A] \rightsquigarrow [\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]} \in [\![\mathsf{Str}_\Sigma[A]]\!]$$

$[\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]}$ determines behavior of w w.r.t. all $\mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ terms:

$$w \in \mathcal{L}(t) \iff t\,\overline{w} \to_\beta^* \mathtt{true} \underbrace{\iff}_{\text{assuming } [\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]} [\![t\,\overline{w}]\!] = [\![t]\!]([\![\overline{w}]\!]) = [\![\mathtt{true}]\!]$$

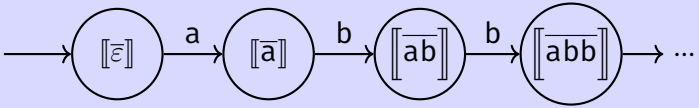
Goal: to decide $\mathcal{L}(t)$, compute $w \mapsto [\![\overline{w}]\!]$ in some denotational model.

Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A and any simply typed λ-term $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$, the language $\mathcal{L}(t) = \{w \in \Sigma^* \mid t\,\overline{w} \to_\beta^* \mathtt{true}\}$ is regular.

Part 2 of proof.

We use $[\![-]\!] : \mathrm{ST}\lambda\mathrm{C} \to \mathsf{FinSet}$ to build a DFA with states $Q = [\![\mathsf{Str}_\Sigma[A]]\!]$, acceptation as $[\![t]\!](-) = [\![\mathtt{true}]\!]$.



$$w \in \mathcal{L}(t) \iff [\![t]\!]\Big([\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]}\Big) = [\![\mathtt{true}]\!] \iff w \text{ accepted}$$

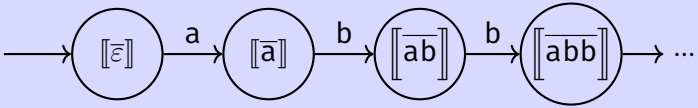
$\longrightarrow$ *semantic evaluation* argument (variant: morphism to monoid $[\![\mathsf{Str}_\Sigma[A]]\!]$) $\square$

Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A and any simply typed λ-term $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$, the language $\mathcal{L}(t) = \{w \in \Sigma^* \mid t\,\overline{w} \to_\beta^* \mathtt{true}\}$ is regular.

Part 2 of proof.

We use $[\![-]\!] : \mathrm{ST}\lambda\mathrm{C} \to \mathsf{FinSet}$ to build a DFA with states $Q = [\![\mathsf{Str}_\Sigma[A]]\!]$, acceptation as $[\![t]\!](-) = [\![\mathtt{true}]\!]$. ($|Q| < \infty$, e.g. $2^{2^{134}}$ when $A = \mathsf{Bool}$ & $|\,[\![o]\!]\,| = 2 = |\Sigma|$)



$$w \in \mathcal{L}(t) \iff [\![t]\!]\Big([\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]}\Big) = [\![\mathtt{true}]\!] \iff w \text{ accepted}$$

$\longrightarrow$ *semantic evaluation* argument (variant: morphism to monoid $[\![\mathsf{Str}_\Sigma[A]]\!]$) $\square$

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

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

There are several canonical ones!

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

("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!

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

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

All **regularity reflecting:** L regular $\implies f^{-1}(L)$ regular

Corollary of the Hillebrand–Kanellakis theorem

Functions defined by $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ in the simply typed λ-calculus are also regularity reflecing.

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

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

There are several canonical ones!

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

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

All **regularity reflecting:** L regular $\implies f^{-1}(L)$ regular

Corollary of the Hillebrand–Kanellakis theorem

Functions defined by $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ in the simply typed λ-calculus are also regularity reflecing.

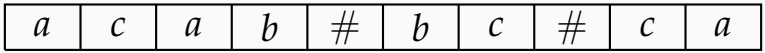
We shall be interested in *regular functions*; many definitions,

such as *streaming string transducers*

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

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

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

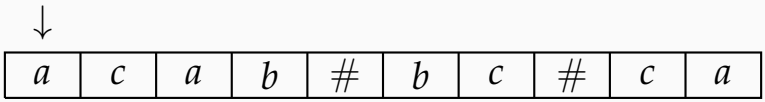


$$X = \varepsilon \qquad Y = \varepsilon$$

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

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

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

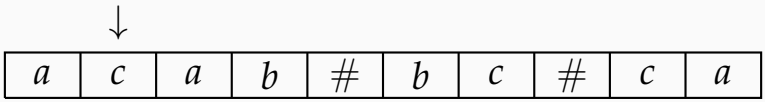


$X = a \qquad Y = \varepsilon$

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

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

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

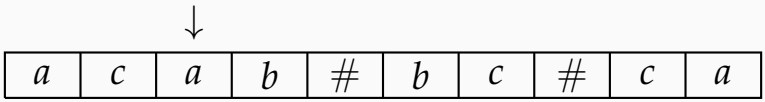


$X = ca \qquad Y = \varepsilon$

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

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

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

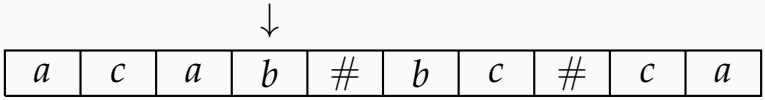


$$X = aca \qquad Y = \varepsilon$$

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

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

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

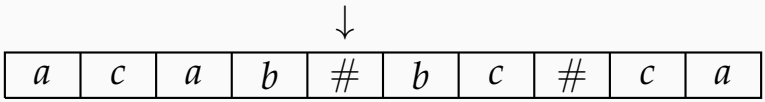


$$X = baca \qquad Y = \varepsilon$$

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

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

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

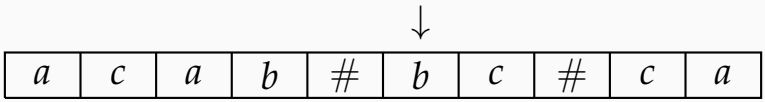


$$X = \varepsilon \qquad Y = baca\#$$

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

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

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

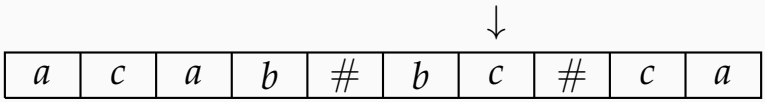


$X = b \qquad Y = baca\#$

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

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

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

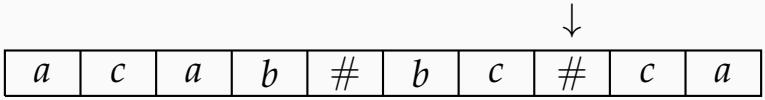


$X = cb \qquad Y = baca\#$

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

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

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

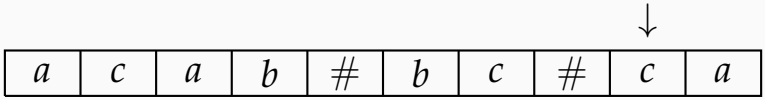


$$X = \varepsilon \qquad Y = baca\#cb\#$$

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

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

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

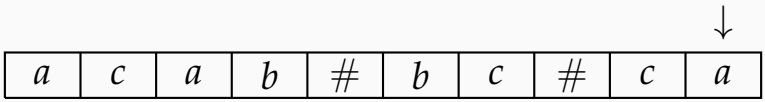


$$X = c \qquad Y = baca\#cb\#$$

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

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

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

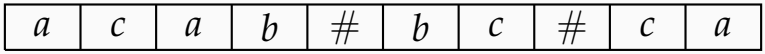


$$X = ac \qquad Y = baca\#cb\#$$

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

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

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

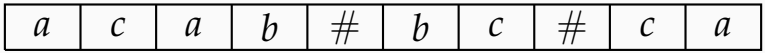


$X = ac \qquad Y = baca\#cb\# \qquad \texttt{mapReverse}(\ldots) = YX = baca\#cb\#ac$

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

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

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



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

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

Linearity

Regular functions = computed by *copyless* streaming string transducers
Restrictions on "copying power": old theme in automaton theory

λ-calculus counterpart: linear types (Girard 1987)

Here, we use the "$\lambda\ell^{\oplus\&}$-calculus" = *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

λ-calculus counterpart: linear types (Girard 1987)

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

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

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

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 (N. & Pradic; proof technique on next slides)

$f : \Gamma^* \to \Sigma^*$ *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 (N. & Pradic; proof technique on next slides)

$f : \Gamma^* \to \Sigma^*$ *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* $\forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$

Works also for regular *tree-to-tree* functions: $\mathsf{Tree} = (o \multimap o \multimap o) \to \cdots \to o$

- Additives $\&/\oplus$ required to get enough expressive power in tree-to-tree case
- String-to-string without $\&/\oplus$: use *affine* types + Krohn-Rhodes decomposition

Proof of $\lambda\ell^{\oplus\&}$-definable $\implies$ regular function

1. By syntactic analysis,[1] turn $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ into a "$\lambda\ell^{\oplus\&}$-transducer"
   - add constant symbols $\varepsilon : o$ and $a : o \multimap o$ for $a \in \Sigma$
   - λ-terms defining a one-way computation:

$$u : A \qquad (t_a : A \multimap A)_{a\in\Gamma} \qquad v : A \multimap o$$

[1] Easy with just $\to$ / $\multimap$ /&, uses focusing with $\otimes/\oplus$.

Proof of $\lambda\ell^{\oplus\&}$-definable $\implies$ regular function

1. By syntactic analysis,[1] turn $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ into a "$\lambda\ell^{\oplus\&}$-transducer"
   - add constant symbols $\varepsilon : o$ and $a : o \multimap o$ for $a \in \Sigma$
   - λ-terms defining a one-way computation:

$$u : A \qquad (t_a : A \multimap A)_{a\in\Gamma} \qquad v : A \multimap o$$

$\lambda\ell^{\oplus\&}$-transducers = variant of *higher-order transducers*

[1] Easy with just $\to$ / $\multimap$ /&, uses focusing with $\otimes/\oplus$.

Proof of $\lambda\ell^{\oplus\&}$-definable $\Longrightarrow$ regular function

1. By syntactic analysis,[1] turn $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ into a "$\lambda\ell^{\oplus\&}$-transducer"
   - add constant symbols $\varepsilon : o$ and $a : o \multimap o$ for $a \in \Sigma$
   - λ-terms defining a one-way computation:

$$u : A \qquad (t_a : A \multimap A)_{a\in\Gamma} \qquad v : A \multimap o$$

$\lambda\ell^{\oplus\&}$-transducers $=$ variant of *higher-order transducers*

2. Compile $\lambda\ell^{\oplus\&}$-transducer to streaming string trans. by semantic evaluation
   - Monoidal closed category tailored for this purpose:
     Dialectica-like bicompletion + functorial automata + string combinatorics

[1] Easy with just $\to$ / $\multimap$ /&, uses focusing with $\otimes/\oplus$.

Alternative proof of affine λ-definable $\Longrightarrow$ regular function

This time **we do not have the additives** $\oplus/\&$

1. By syntactic analysis, turn $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ into an "affine λ-transducer"

Alternative proof of affine λ-definable $\Longrightarrow$ regular function

This time **we do not have the additives** $\oplus/\&$

1. By syntactic analysis, turn $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ into an "affine λ-transducer"
2. Compile affine λ-trans. to $\overbrace{\textit{two-way transducers}}^{\text{next slide}}$ by $\overbrace{\textit{Geometry of Interaction}}^{\text{2 approaches}}$
   - Interaction Abstract Machine [N. & Vanoni '25]
   - categorical GoI [Pradic & Price '24], following earlier $\mathsf{Int}(\mathcal{C})$ vs automata works [Hines '03, Katsumata '08]

Alternative proof of affine λ-definable $\Longrightarrow$ regular function

This time **we do not have the additives** $\oplus/\&$

1. By syntactic analysis, turn $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ into an "affine λ-transducer"
2. Compile affine λ-trans. to $\overbrace{\textit{two-way transducers}}^{\text{next slide}}$ by $\overbrace{\textit{Geometry of Interaction}}^{\text{2 approaches}}$
   - Interaction Abstract Machine [N. & Vanoni '25]
   - categorical GoI [Pradic & Price '24], following earlier $\mathsf{Int}(\mathcal{C})$ vs automata works [Hines '03, Katsumata '08]

Over trees: affine λ-trans. $\rightsquigarrow$ tree-walking transducers

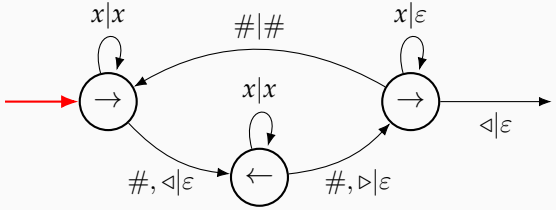
Theorem (Bojańczyk & Colcombet 2005)

Tree-walking automata $\subsetneq$ regular tree languages

Corollary: affine λ-transducers $\subsetneq$ regular tree functions

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



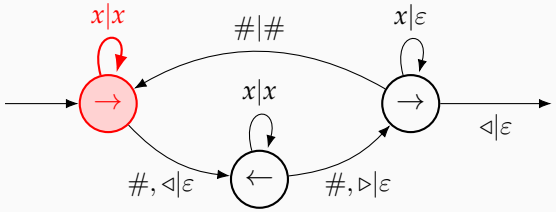
$(x \in \{a, b, c\})$

▷	a	b	c	#	b	a	c	#	c	b	◁

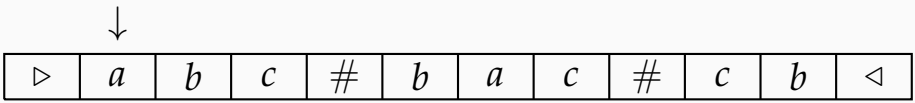
Output:

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



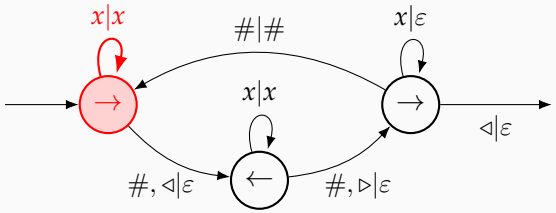
$(x \in \{a, b, c\})$



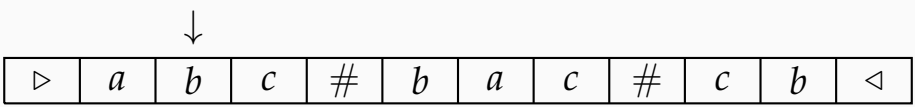
Output:

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



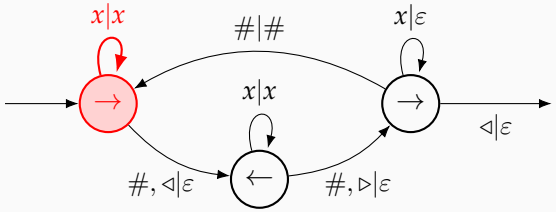
$(x \in \{a, b, c\})$



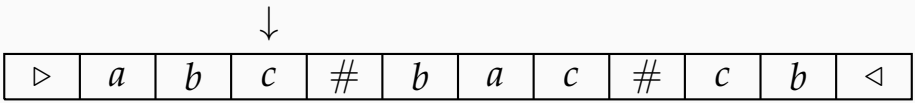
Output: a

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



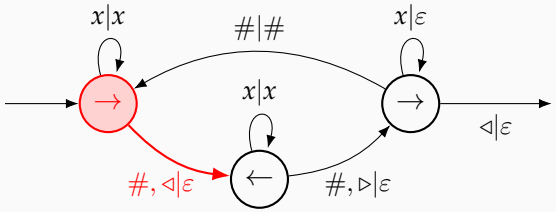
$(x \in \{a, b, c\})$



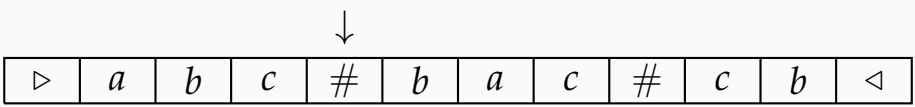
Output: ab

Two-way transducers characterize regular functions

Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



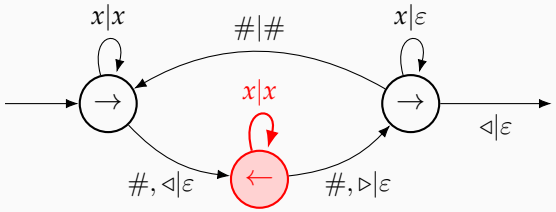
$(x \in \{a, b, c\})$



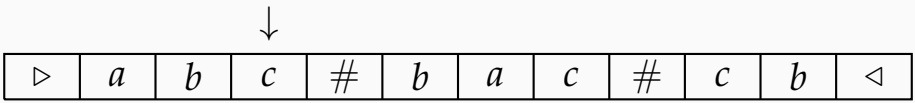
Output: abc

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



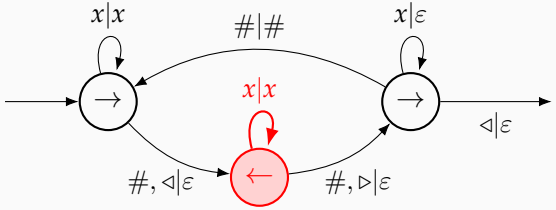
$(x \in \{a, b, c\})$



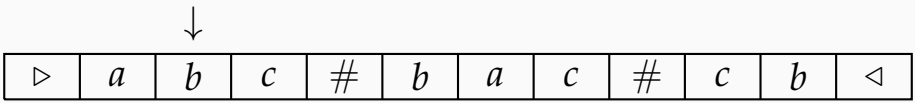
Output: abc

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



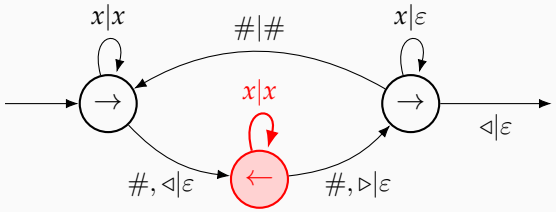
$(x \in \{a, b, c\})$



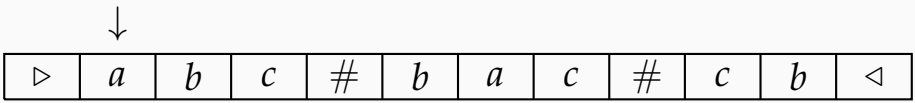
Output: *abcc*

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



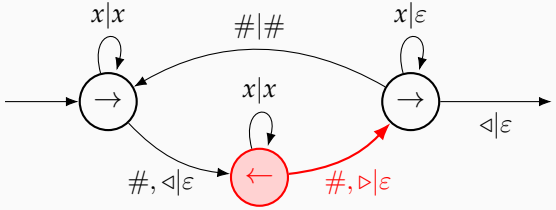
$(x \in \{a, b, c\})$



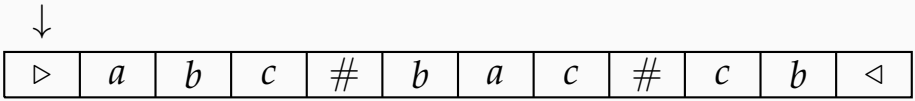
Output: *abccb*

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



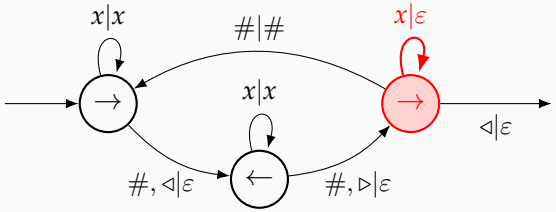
$(x \in \{a, b, c\})$



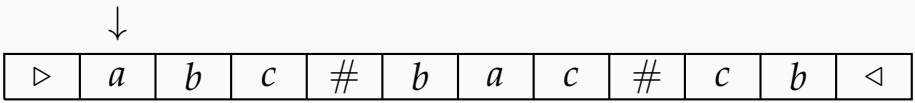
Output: *abccba*

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



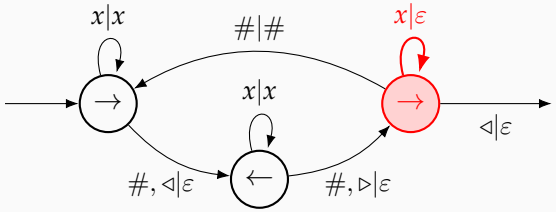
$(x \in \{a, b, c\})$



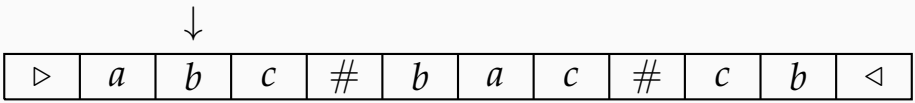
Output: *abccba*

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



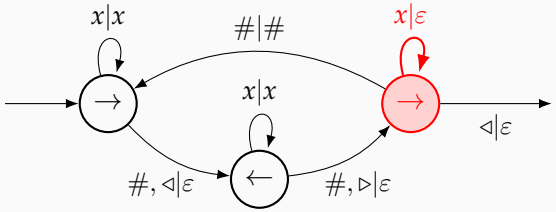
$(x \in \{a, b, c\})$



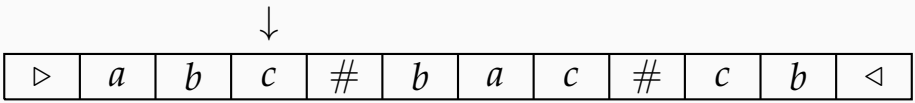
Output: *abccba*

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



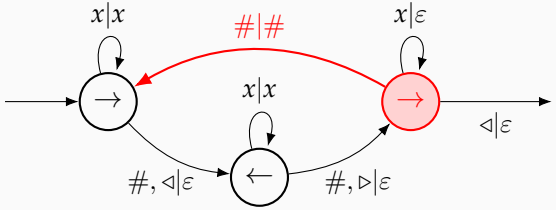
$(x \in \{a, b, c\})$



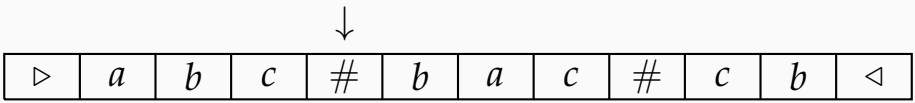
Output: *abccbaa*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



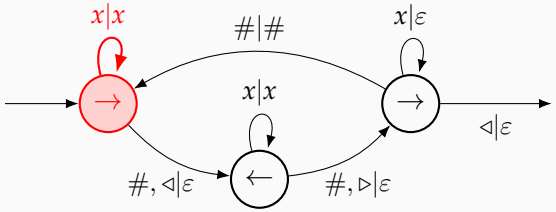
$(x \in \{a, b, c\})$



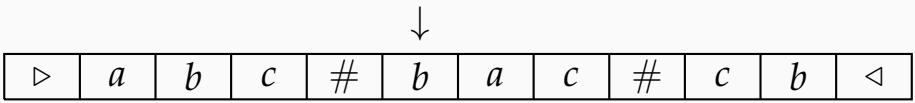
Output: *abccba*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



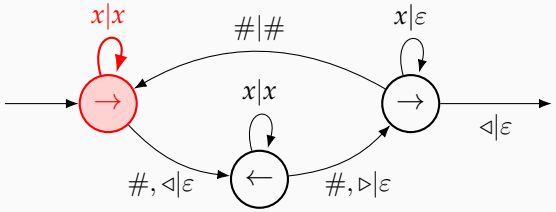
$(x \in \{a, b, c\})$



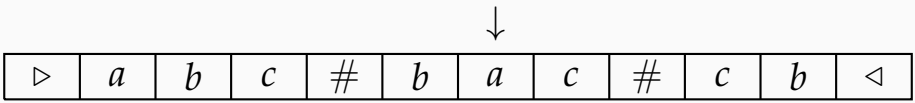
Output: $abccba\#$

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



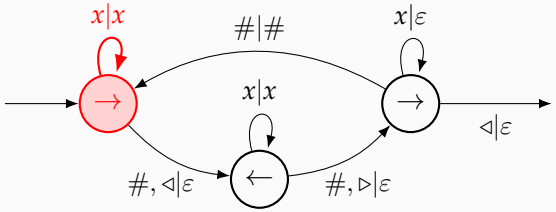
$(x \in \{a, b, c\})$



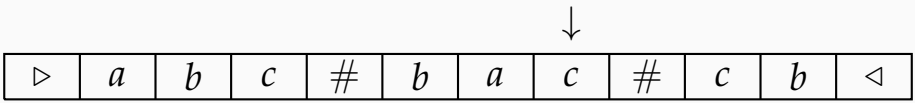
Output: $abccba\#b$

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



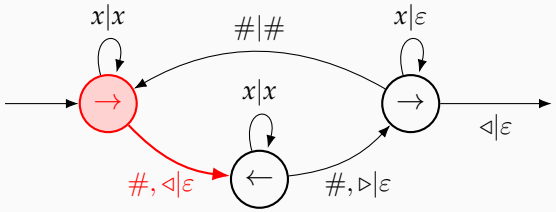
$(x \in \{a, b, c\})$



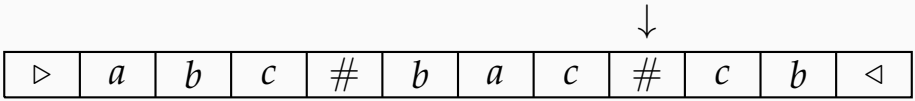
Output: $abccba\#ba$

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



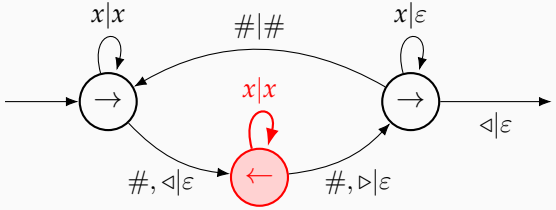
$(x \in \{a, b, c\})$



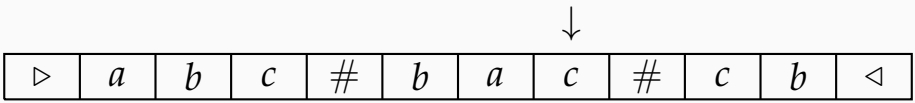
Output: $abccba\#bac$

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



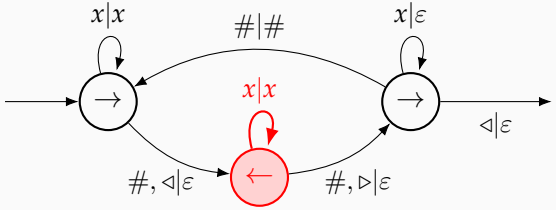
$(x \in \{a, b, c\})$



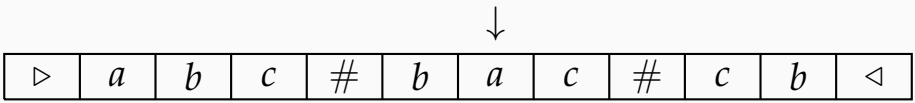
Output: $abccba\#bac$

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



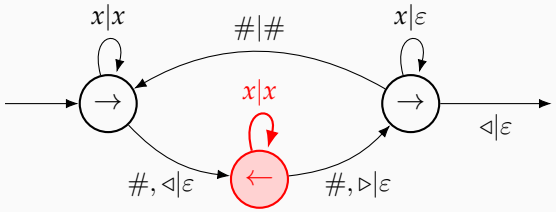
$(x \in \{a, b, c\})$



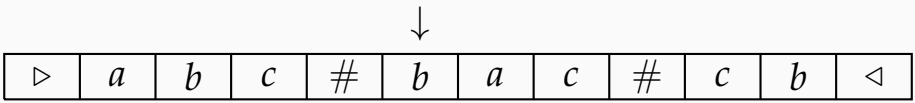
Output: *abccba#bacc*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



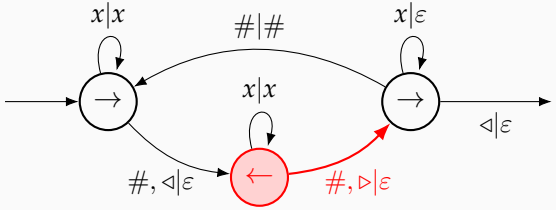
$(x \in \{a, b, c\})$



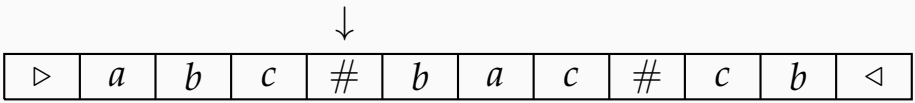
Output: *abccba#bacca*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



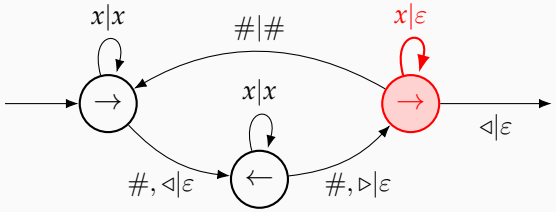
$(x \in \{a, b, c\})$



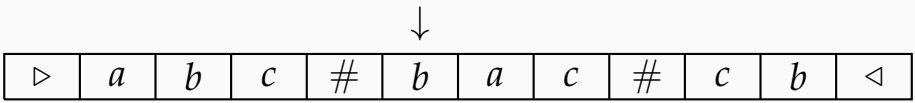
Output: *abccba#baccab*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



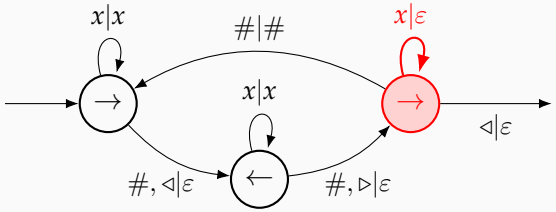
$(x \in \{a, b, c\})$



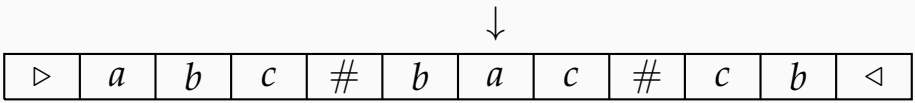
Output: *abccba#baccab*

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



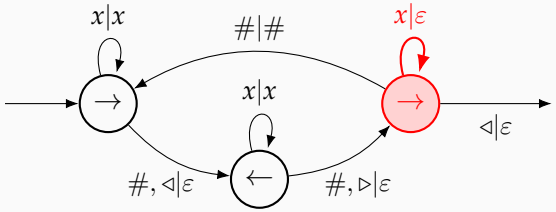
$(x \in \{a, b, c\})$



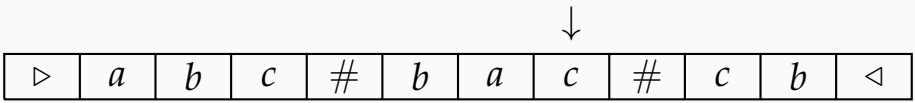
Output: *abccba#baccab*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



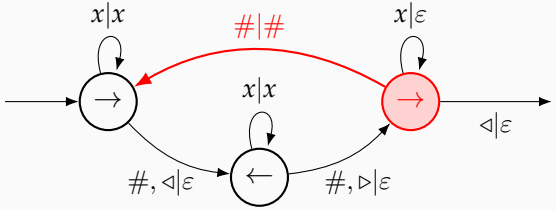
$(x \in \{a, b, c\})$



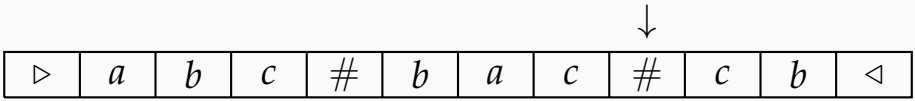
Output: *abccba#baccab*

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



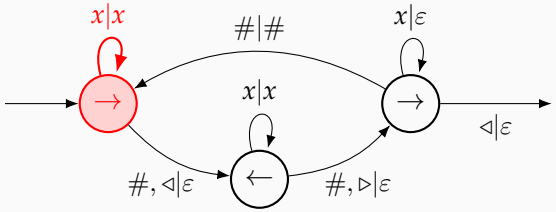
$(x \in \{a, b, c\})$



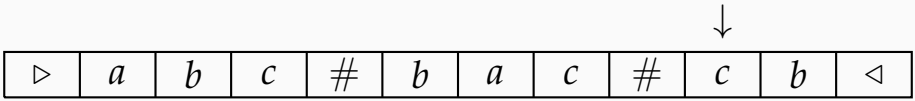
Output: *abccba#baccab*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



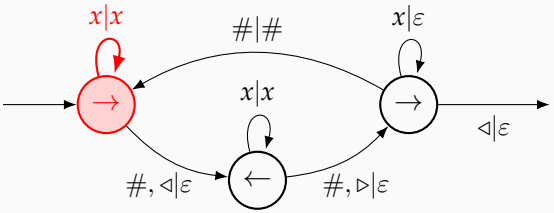
$(x \in \{a, b, c\})$



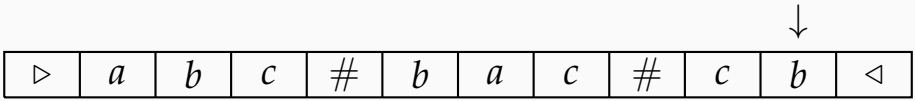
Output: $abccba\#baccab\#$

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



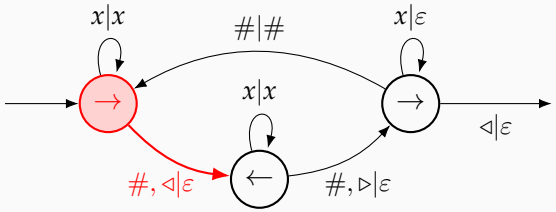
$(x \in \{a, b, c\})$



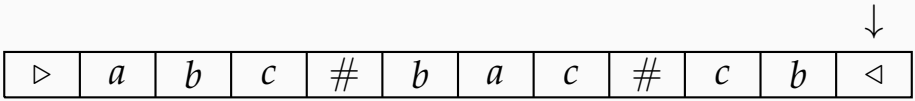
Output: $abccba\#baccab\#c$

Two-way transducers characterize regular functions

Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



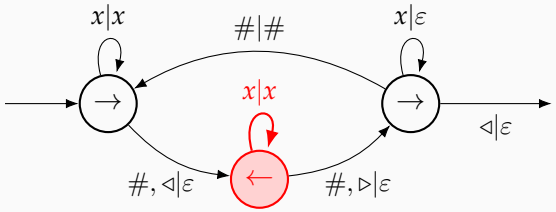
$(x \in \{a, b, c\})$



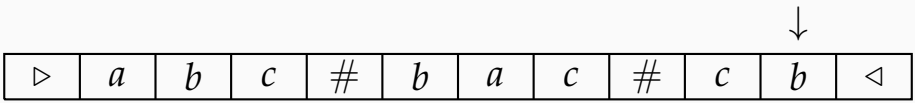
Output: *abccba#baccab#cb*

Two-way transducers characterize regular functions

Example: $w_1\#\ldots\#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\#\ldots\#w_n \cdot \texttt{reverse}(w_n)$



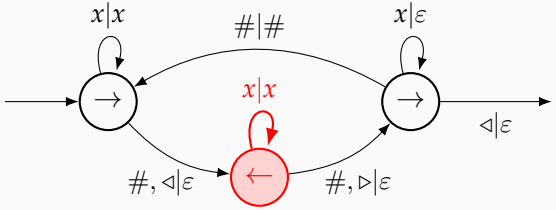
$(x \in \{a, b, c\})$



Output: $abccba\#baccab\#cb$

Two-way transducers characterize regular functions

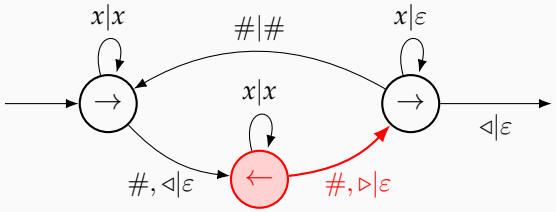
Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



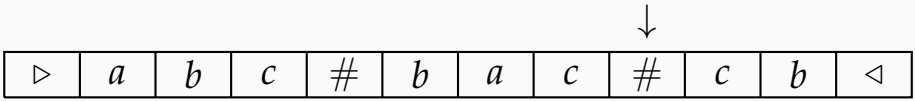
Output: *abccba#baccab#cbb*

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



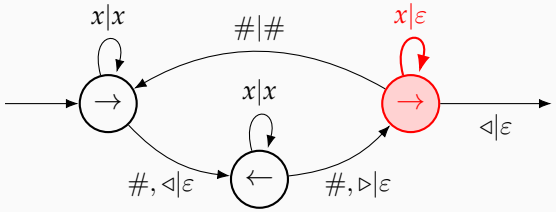
$(x \in \{a, b, c\})$



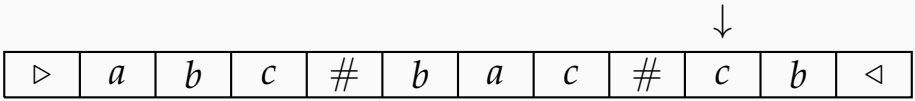
Output: *abccba#baccab#cbbc*

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



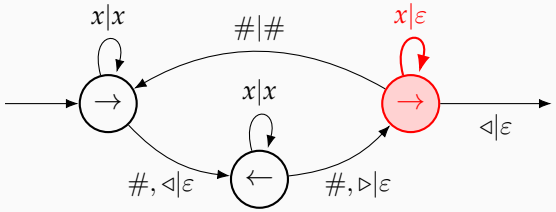
$(x \in \{a, b, c\})$



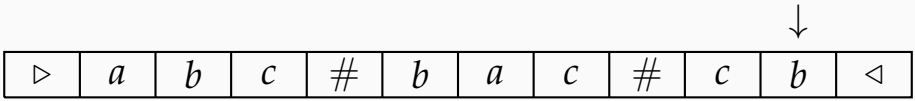
Output: $abccba\#baccab\#cbbc$

Two-way transducers characterize regular functions

Example: $w_1\# \ldots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \ldots \#w_n \cdot \texttt{reverse}(w_n)$



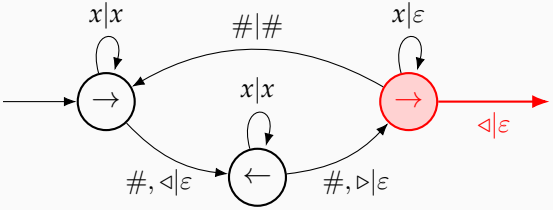
$(x \in \{a, b, c\})$



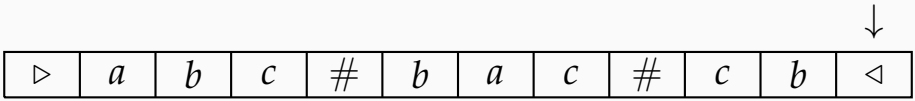
Output: $abccba\#baccab\#cbbc$

Two-way transducers characterize regular functions

Example: $w_1\# \dots \# w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \# w_n \cdot \texttt{reverse}(w_n)$



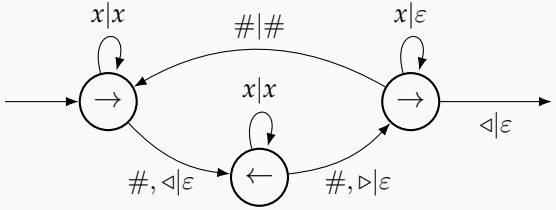
$(x \in \{a, b, c\})$



Output: *abccba#baccab#cbbc*

Two-way transducers characterize regular functions

Example: $w_1\# \dots \#w_n \longmapsto w_1 \cdot \texttt{reverse}(w_1)\# \dots \#w_n \cdot \texttt{reverse}(w_n)$



$(x \in \{a, b, c\})$

▷	a	b	c	#	b	a	c	#	c	b	◁

Output: $abccba\#baccab\#cbbc$

λ-transducers with regular lookaround

Corollary of Bojańczyk & Colcombet 2005

Affine tree-to-tree λ-transducers $\subsetneq$ regular tree functions.

- Previously: to get $=$, use additives $\oplus/\&$

λ-transducers with regular lookaround

Corollary of Bojańczyk & Colcombet 2005

Affine tree-to-tree λ-transducers $\subsetneq$ regular tree functions.

- Previously: to get $=$, use additives $\oplus/\&$
- Alternatively: query regular properties of input during computation

Theorem (Gallot, Lemay & Salvati 2021)

Affine tree-to-tree λ-trans. with regular lookaround $=$ *regular tree functions.*

λ-transducers with regular lookaround

Corollary of Bojańczyk & Colcombet 2005

Affine tree-to-tree λ-transducers $\subsetneq$ regular tree functions.

- Previously: to get $=$, use additives $\oplus/\&$
- Alternatively: query regular properties of input during computation

Theorem (Gallot, Lemay & Salvati 2021)

Affine tree-to-tree λ-trans. with regular lookaround $=$ *regular tree functions.*

The *exponential modality* '!' of linear logic makes a type duplicable

Slightly non-linear λ-transducers with regular lookaround [N. & Vanoni 2025]

Relating λ-transducers of low !-depth to automata models,
e.g. "invisible pebble tree transducers"

Proofs: via Interaction Abstract Machine

Higher-order transducers vs implicit automata

Pros and cons of λ-transducers:

- Easy to add automata-theoretic features, e.g. regular lookaround
- Only traverse their input once

Higher-order transducers vs implicit automata

Pros and cons of λ-transducers:

- Easy to add automata-theoretic features, e.g. regular lookaround
- Only traverse their input once — implicit automata can relax this constraint

Theorem (N. & Pradic — note the $\to$ instead of $\multimap$)

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

Higher-order transducers vs implicit automata

Pros and cons of λ-transducers:

- Easy to add automata-theoretic features, e.g. regular lookaround
- Only traverse their input once — implicit automata can relax this constraint

Theorem (N. & Pradic — note the $\to$ instead of $\multimap$)

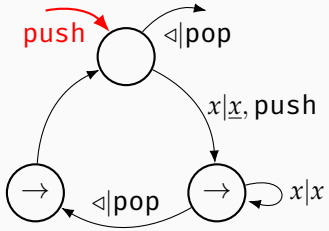
$$f : \Gamma^* \to \Sigma^* \textit{ is } \text{comparison-free polyregular}$$
$$\iff$$
$$\exists\, a \text{ purely linear } \textit{type } A \textit{ and } t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma \textit{ in the } \lambda\ell^{\oplus\&}\textit{-calculus}$$
$$\textit{such that } \forall w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

$\rightsquigarrow$ *Discovery* of the "comparison-free" subclass of Bojańczyk's *polyregular functions*, and of its good properties (e.g. closure under composition)

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



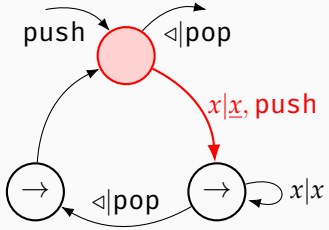
▷	a	b	c	◁

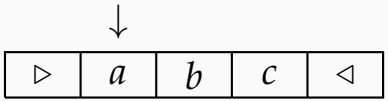
output =

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



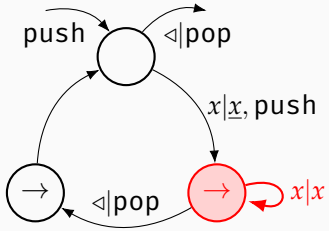


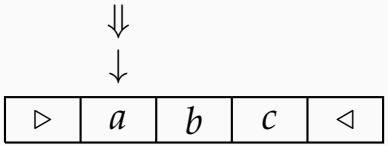
output =

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: “comparison-free squaring”

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



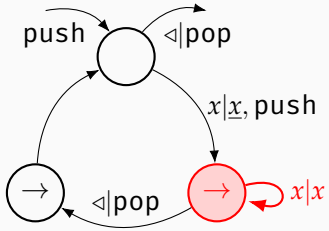


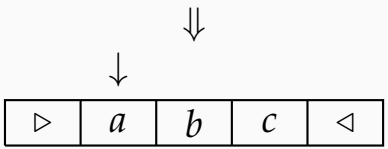
output $= \underline{a}$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



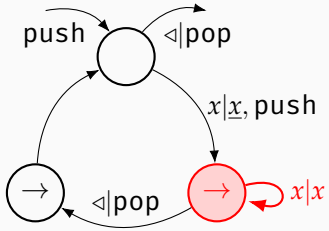


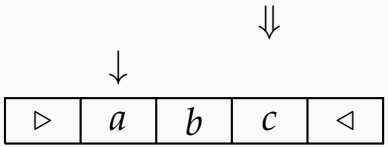
output $= \underline{a}a$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



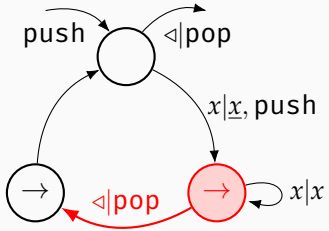


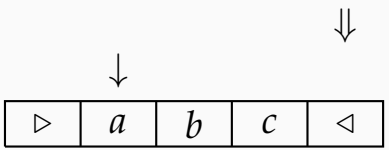
output $= \underline{a}ab$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



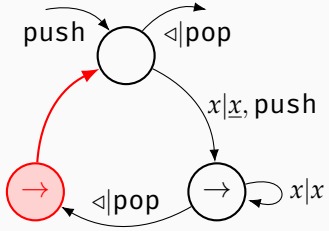


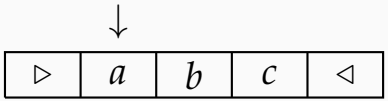
output $= \underline{a}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



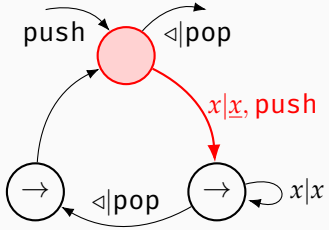


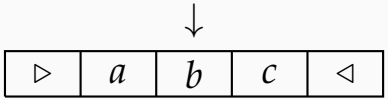
output $= \underline{a}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



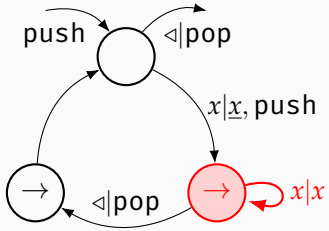


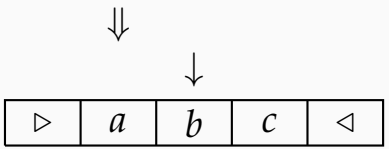
output $= \underline{a}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: “comparison-free squaring”

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



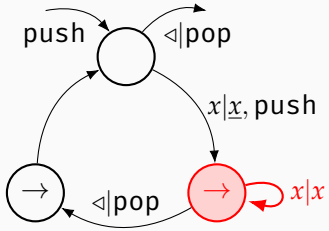


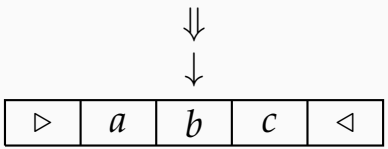
output $= \underline{a}abc\underline{b}$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



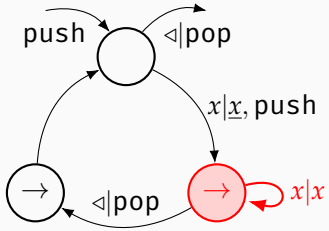


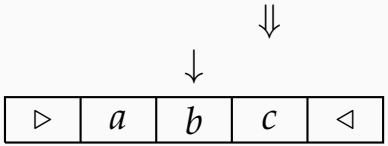
output $= \underline{a}abc\underline{b}a$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$

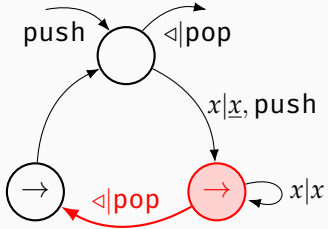




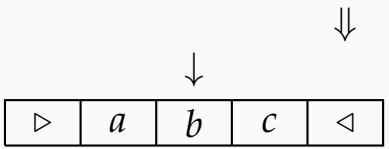
output $= \underline{a}abc\underline{b}ab$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"



$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$

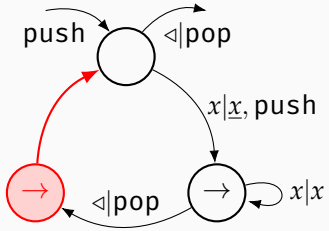


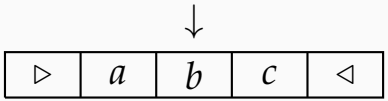
output $= \underline{a}abc\underline{b}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



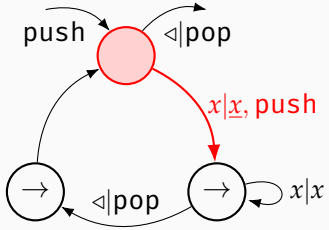


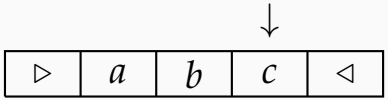
output $= \underline{a}abc\underline{b}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



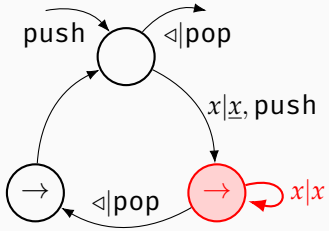


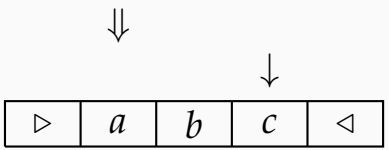
output $= \underline{a}abc\underline{b}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



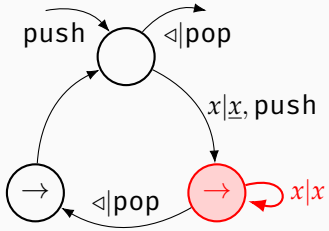


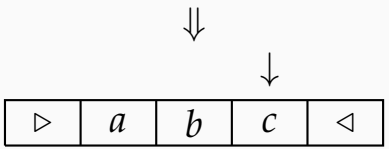
output $= \underline{a}abc\underline{b}abc\underline{c}$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



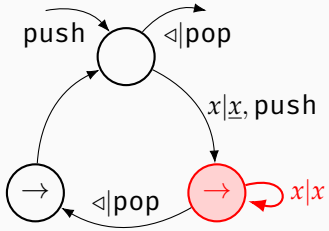


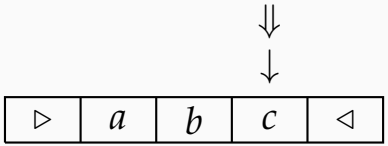
output $= \underline{a}abc\underline{b}abc\underline{c}a$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



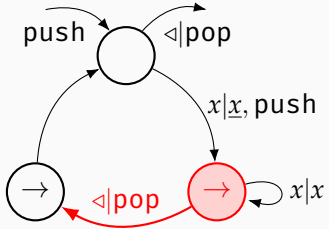


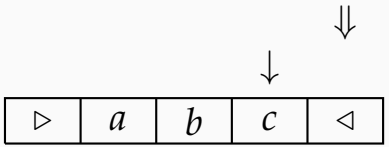
output $= \underline{a}abc\underline{b}abcc\underline{a}b$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



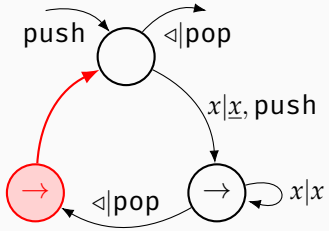


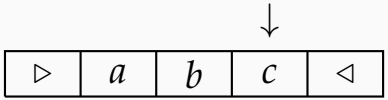
output $= \underline{a}abc\underline{b}abc\underline{c}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



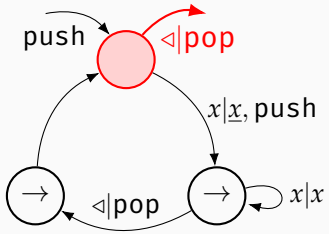


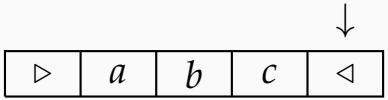
output $= \underline{a}abc\underline{b}abc\underline{c}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



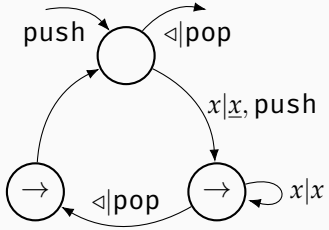


output $= \underline{a}abc\underline{b}abc\underline{c}abc$

Comparison-free polyregular functions, a.k.a. polyblind functions

Example: "comparison-free squaring"

$\texttt{cfsquaring}(abb) = \underline{a}abb\underline{b}abb\underline{b}abb$



▷	a	b	c	◁

output $= \underline{a}abc\underline{b}abc\underline{c}abc$

Non-misery of implicit automata??

Good news: we've worked with "natural" typed λ-calculi for now

Non-misery of implicit automata??

Good news: we've worked with "natural" typed λ-calculi for now

From Mazza's habilitation thesis (2017):

- *"the characterization results* [*in implicit computational complexity*] *are composed of two parts bearing little relevance with the question of lower bounds: a completeness part, which is generally a programming exercise of limited theoretical interest* [...]"
  in implicit automata: sometimes use non-trivial theory (Krohn–Rhodes)

Non-misery of implicit automata??

Good news: we've worked with "natural" typed λ-calculi for now

From Mazza's habilitation thesis (2017):

- *"the characterization results* [*in implicit computational complexity*] *are composed of two parts bearing little relevance with the question of lower bounds: a completeness part, which is generally a programming exercise of limited theoretical interest* [...]"
  in implicit automata: sometimes use non-trivial theory (Krohn–Rhodes)
- *"for the time being, there is no hope of making progress in structural complexity theory by use of tools from logic and programming languages theory"*
  we've seen that implicit automata can suggest objects of interest for "pure" transducer theory, and help prove their properties

Non-misery of implicit automata??

Good news: we've worked with "natural" typed λ-calculi for now

From Mazza's habilitation thesis (2017):

- *"the characterization results* [*in implicit computational complexity*] *are composed of two parts bearing little relevance with the question of lower bounds: a completeness part, which is generally a programming exercise of limited theoretical interest* [...]"
  in implicit automata: sometimes use non-trivial theory (Krohn–Rhodes)
- *"for the time being, there is no hope of making progress in structural complexity theory by use of tools from logic and programming languages theory"*
  we've seen that implicit automata can suggest objects of interest for "pure" transducer theory, and help prove their properties

Next: ideas from λ-calculus help solve an open problem in automata

The power of additive branching

Hennie machines = *bounded-visit* two-way transducers that can write on input cells

Theorem (Dartois, N., Peyrat '26)

The following define the same functions:

- $t : \mathsf{Tree}^{\otimes}[A] \multimap \mathsf{Tree}^{\&}$, A *$\to$-free*
- *tree-to-tree* Hennie machines.

$$\mathsf{Tree}^{\otimes} = (o \multimap o \multimap o) \to \cdots \to o$$

$$\mathsf{Tree}^{\&} = ((o \,\&\, o) \multimap o) \to \cdots \to o$$

The power of additive branching

Hennie machines = *bounded-visit* two-way transducers that can write on input cells

Theorem (Dartois, N., Peyrat '26)

The following define the same functions:

- $t : \mathsf{Tree}^{\otimes}[A] \multimap \mathsf{Tree}^{\&}, A \to$*-free*
- *tree-to-tree* Hennie machines.

$\mathsf{Tree}^{\otimes} = (o \multimap o \multimap o) \to \cdots \to o$

$\mathsf{Tree}^{\&} = ((o \,\&\, o) \multimap o) \to \cdots \to o$

Theorem (Clairambault, Murawski '19)

The following define the same infinite trees:

- *affine* higher-order recursion schemes *with additive branching;*
- *tree stack automata.*

Morally same result, via close analogies

The power of additive branching

Hennie machines = *bounded-visit* two-way transducers that can write on input cells

Theorem (Dartois, N., Peyrat '26)

The following define the same functions:

- $t : \mathsf{Tree}^{\otimes}[A] \multimap \mathsf{Tree}^{\&}$, $A \to$*-free*
- *tree-to-tree* Hennie machines.

$\mathsf{Tree}^{\otimes} = (o \multimap o \multimap o) \to \cdots \to o$

$\mathsf{Tree}^{\&} = ((o \,\&\, o) \multimap o) \to \cdots \to o$

Theorem (Clairambault, Murawski '19)

The following define the same infinite trees:

- *affine* higher-order recursion schemes *with additive branching;*
- *tree stack automata.*

Morally same result, via close analogies

Proofs: continuation-passing style encodings / memoryful GoI with additives

one is a straightfoward transposition of the other

Consequences for automata theory

Theorem (Dartois, N., Peyrat '26)

The following define the same functions:

- $t : \mathsf{Tree}^{\otimes}[A] \multimap \mathsf{Tree}^{\&}$, $A \to$*-free*
- *tree-to-tree* Hennie machines.

Corollary

Tree-to-tree Hennie machines are regularity reflecting*: L regular $\implies f^{-1}(L)$ regular*

No known proof of this fact that avoids "higher-order" aspects!

Consequences for automata theory

Theorem (Dartois, N., Peyrat '26)

The following define the same functions:

- $t : \mathsf{Tree}^{\otimes}[A] \multimap \mathsf{Tree}^{\&}$, $A \to$*-free*
- *tree-to-tree* Hennie machines.

Corollary

Tree-to-tree Hennie machines are regularity reflecting*: L regular $\implies f^{-1}(L)$ regular*

No known proof of this fact that avoids "higher-order" aspects!

- Easy application: alternating linear-time single-tape Turing machines recognize only regular languages

Consequences for automata theory

Theorem (Dartois, N., Peyrat '26)

The following define the same functions:

- $t : \mathsf{Tree}^{\otimes}[A] \multimap \mathsf{Tree}^{\&}$, $A \to$*-free*
- *tree-to-tree* Hennie machines.

Corollary

Tree-to-tree Hennie machines are regularity reflecting*: L regular $\implies f^{-1}(L)$ regular*

No known proof of this fact that avoids "higher-order" aspects!

- Easy application: alternating linear-time single-tape Turing machines recognize only regular languages
- Hard application:

Theorem (Colcombet, Lhote & Ohlmann '26 — first conjectured by Bárány '07)

Every automatic ω-word has a decidable MSO theory.

More results

- *non-commutative* affine λ-calculus $\rightsquigarrow$ star-free languages [N. & Pradic '20]
- simply typed (light|parsimonious) affine λ-calculus $\rightsquigarrow$ polyregular functions [Wojciech Paupa's master thesis, Univ. Warsaw, to appear]

More results

- *non-commutative* affine λ-calculus $\rightsquigarrow$ star-free languages [N. & Pradic '20]
- simply typed (light|parsimonious) affine λ-calculus $\rightsquigarrow$ polyregular functions
  [Wojciech Paupa's master thesis, Univ. Warsaw, to appear]
- *safety* condition of higher-order recursion schemes, implicit in older works

$$\text{safe order-}k\text{ transducers} = k\text{-iterated pushdown transducers}$$

[Engelfriet & Vogler 1988] $$= (\text{order-1 a.k.a. macro tree transducers})^k$$

More results

- *non-commutative* affine λ-calculus $\rightsquigarrow$ star-free languages [N. & Pradic '20]
- simply typed (light|parsimonious) affine λ-calculus $\rightsquigarrow$ polyregular functions
  [Wojciech Paupa's master thesis, Univ. Warsaw, to appear]
- *safety* condition of higher-order recursion schemes, implicit in older works

$$\text{safe order-}k\text{ transducers} = k\text{-iterated pushdown transducers}$$

$$\text{[Engelfriet \& Vogler 1988]} \qquad = (\text{order-1 a.k.a. macro tree transducers})^k$$

Simply typed λ-calculus / unsafe higher-order transducers: open questions

- Reflection of regular cost functions?
- Deciding asymptotic growth? Polynomial growth implies polyregular?

More results

- *non-commutative* affine λ-calculus $\rightsquigarrow$ star-free languages [N. & Pradic '20]
- simply typed (light|parsimonious) affine λ-calculus $\rightsquigarrow$ polyregular functions
  [Wojciech Paupa's master thesis, Univ. Warsaw, to appear]
- *safety* condition of higher-order recursion schemes, implicit in older works

$$\text{safe order-}k\text{ transducers} = k\text{-iterated pushdown transducers}$$

$$[\text{Engelfriet \& Vogler 1988}] \qquad = (\text{order-1 a.k.a. macro tree transducers})^k$$

Simply typed λ-calculus / unsafe higher-order transducers: open questions

- Reflection of regular cost functions?
- Deciding asymptotic growth? Polynomial growth implies polyregular?
- Study of order-k output languages (motivation: [Kiefer, N., Pradic '23])
  $\rightsquigarrow$ use a $\{0, 1, \infty\}$-graded Taylor expansion with remainder of λ-terms?

- *non-commutative* affine λ-calculus $\rightsquigarrow$ star-free languages [N. & Pradic '20]
- simply typed (light|parsimonious) affine λ-calculus $\rightsquigarrow$ polyregular functions [Wojciech Paupa's master thesis, Univ. Warsaw, to appear]
- *safety* condition of higher-order recursion schemes, implicit in older works

safe order-k transducers $=$ k-iterated pushdown transducers

[Engelfriet & Vogler 1988] $=$ (order-1 a.k.a. macro tree transducers)^k

Simply typed λ-calculus / unsafe higher-order transducers: open questions

- Reflection of regular cost functions?
- Deciding asymptotic growth? Polynomial growth implies polyregular?
- Study of order-k output languages (motivation: [Kiefer, N., Pradic '23])
  $\rightsquigarrow$ use a $\{0, 1, \infty\}$-graded Taylor expansion with remainder of λ-terms?