Automates implicites en λ-calcul typé

Lê Thành Dũng (Tito) Nguyễn — nltd@nguyentito.eu
travaux en collaboration avec Pierre Pradic

Séminaire Logique de la Programmation
Institut de Mathématiques de Marseille, jeudi 3 mars 2022

Quelques motivations provenant du λ-calcul

Considérons le *λ-calcul simplement typé* (supposé connu).

C'est un langage de programmation, et il termine, donc non Turing-complet...

Quelques motivations provenant du λ-calcul

Considérons le *λ-calcul simplement typé* (supposé connu).

C'est un langage de programmation, et il termine, donc non Turing-complet...
$\longrightarrow$ question typique de « pouvoir expressif » : *que calcule-t-il ?*
Quelques résultats connus, par ex :

Théorème (Schwichtenberg 1975)

Les fonctions $\mathbb{N}^k \to \mathbb{N}$ définissables par des λ-termes t : $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ simplement typés sont les polynômes étendus (*engendrés par 0, 1, $+$, $\times$, $\mathtt{id}$, $\mathtt{ifzero}$*).

où Nat est le type des *entiers de Church*

Quelques motivations provenant du λ-calcul

Considérons le *λ-calcul simplement typé* (supposé connu).

C'est un langage de programmation, et il termine, donc non Turing-complet...
$\longrightarrow$ question typique de « pouvoir expressif » : *que calcule-t-il ?*
Quelques résultats connus, par ex :

Théorème (Schwichtenberg 1975)

Les fonctions $\mathbb{N}^k \to \mathbb{N}$ définissables par des λ-termes t : $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ simplement typés sont les polynômes étendus (*engendrés par 0, 1, $+$, $\times$, $\mathtt{id}$, $\mathtt{ifzero}$*).

où Nat est le type des *entiers de Church*

Mais certaines questions ouvertes n'ont pas de réponse satisfaisante...

Fonctions sur les entiers en λ-calcul simplement typé

Rappel : le type des entiers de Church est $\mathsf{Nat} = (o \to o) \to o \to o$

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

- pour $n \in \mathbb{N}$, on a $\overline{n} : \mathsf{Nat}$
- réciproquement $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975 : $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = polynômes étendus

Fonctions sur les entiers en λ-calcul simplement typé

Rappel : le type des entiers de Church est $\mathsf{Nat} = (o \to o) \to o \to o$

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

- pour $n \in \mathbb{N}$, on a $\overline{n} : \mathsf{Nat}$ donc $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- réciproquement $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975 : $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = polynômes étendus

Fonctions sur les entiers en λ-calcul simplement typé

Rappel : le type des entiers de Church est $\mathsf{Nat} = (o \to o) \to o \to o$

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

- pour $n \in \mathbb{N}$, on a $\overline{n} : \mathsf{Nat}$ donc $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- réciproquement $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975 : $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = polynômes étendus

Problème ouvert

Quelles fonctions $f : \mathbb{N} \to \mathbb{N}$ peuvent être définies par des λ-termes simplement typés $t : \mathsf{Nat}[A] \to \mathsf{Nat}$ (où A peut être choisi en fonction de f) ?

Fonctions sur les entiers en λ-calcul simplement typé

Rappel : le type des entiers de Church est $\mathsf{Nat} = (o \to o) \to o \to o$

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

- pour $n \in \mathbb{N}$, on a $\overline{n} : \mathsf{Nat}$ donc $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- réciproquement $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

Schwichtenberg 1975 : $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = polynômes étendus

Problème ouvert

Quelles fonctions $f : \mathbb{N} \to \mathbb{N}$ peuvent être définies par des λ-termes simplement typés $t : \mathsf{Nat}[A] \to \mathsf{Nat}$ (où A peut être choisi en fonction de f) ?

Pour $\mathsf{Nat}[A] \to \mathsf{Bool}$, on a une caractérisation satisfaisante !

Langages définissables en λ-calcul simplement typé

Codages de Church des mots binaires [Böhm & Berarducci 1985]

$\simeq$ `fold_right` sur une liste de lettres (s'étend à tout alphabet ; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$) :

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

Langages définissables en λ-calcul simplement typé

Codages de Church des mots binaires [Böhm & Berarducci 1985]

$\simeq$ `fold_right` sur une liste de lettres (s'étend à tout alphabet ; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$) :

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

λ-termes simplement typés $\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \to \mathsf{Bool}$ $\quad\rightsquigarrow\quad$ **langages** $L \subseteq \{\texttt{0},\texttt{1}\}^*$

Langages définissables en λ-calcul simplement typé

Codages de Church des mots binaires [Böhm & Berarducci 1985]

$\simeq$ `fold_right` sur une liste de lettres (s'étend à tout alphabet ; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$) :

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

λ-termes simplement typés $\mathsf{Str}_{\{0,1\}}[A] \to \mathsf{Bool}$ $\quad\rightsquigarrow\quad$ **langages** $L \subseteq \{0,1\}^*$

Exemple : $t = \lambda s.\ s\ \texttt{id}\ \texttt{not}\ \texttt{true} : \mathsf{Str}_{\{0,1\}}[\mathsf{Bool}] \to \mathsf{Bool}$ (nombre pair de 1)

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

Langages définissables en λ-calcul simplement typé

Codages de Church des mots binaires [Böhm & Berarducci 1985]

$\simeq$ `fold_right` sur une liste de lettres (s'étend à tout alphabet ; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$) :

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

λ-termes simplement typés $\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \to \mathsf{Bool} \quad \rightsquigarrow \quad$ **langages** $L \subseteq \{\texttt{0}, \texttt{1}\}^*$

Exemple : $t = \lambda s.\ s\ \texttt{id}\ \texttt{not}\ \texttt{true} : \mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[\mathsf{Bool}] \to \mathsf{Bool}$ (nombre pair de 1)

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

Théorème (Hillebrand & Kanellakis 1996)

Tous les langages rationnels*, et seulement ceux-ci, peuvent être définis de la sorte.*

Un peu de contexte : la complexité implicite

Gabarit pour un théorème reliant « structure » et « pouvoir »

Les langages/functions calculées par des programmes de type T dans le langage de programmation $\mathcal{P}$ sont exactement ceux/celles dans la classe $\mathcal{C}$.

(dichotomie empruntée à Abramsky, cf. par ex. workshop *Structure meets Power*)

- Hillebrand & Kanellakis : $\mathcal{P} = \lambda$-c. simplement typé, $\mathcal{C} =$ langages rationnels
  - Bonne nouvelle : contrairement aux « polynômes étendus »,
    un objet central d'un autre domaine de l'informatique, la *théorie des automates*
  - La définition sera rappelée bientôt

Un peu de contexte : la complexité implicite

Gabarit pour un théorème reliant « structure » et « pouvoir »

Les langages/functions calculées par des programmes de type T dans le langage de programmation $\mathcal{P}$ sont exactement ceux/celles dans la classe $\mathcal{C}$.

(dichotomie empruntée à Abramsky, cf. par ex. workshop *Structure meets Power*)

- Hillebrand & Kanellakis : $\mathcal{P} = \lambda$-c. simplement typé, $\mathcal{C}$ = langages rationnels
  - Bonne nouvelle : contrairement aux « polynômes étendus », un objet central d'un autre domaine de l'informatique, la *théorie des automates*
  - La définition sera rappelée bientôt
- *Complexité implicite* : $\mathcal{C}$ est une classe de complexité algorithmique (P, NP, ...)
  - un domaine actif de recherche depuis les années 1990 (cf. HDR de Péchoux)
  - Exemple historique (Girard) : $\mathcal{P}$ = logique linéaire allégée, $\mathcal{C} = \mathsf{P}$

Un peu de contexte : la complexité implicite

Gabarit pour un théorème reliant « structure » et « pouvoir »

Les langages/functions calculées par des programmes de type T dans le langage de programmation $\mathcal{P}$ sont exactement ceux/celles dans la classe $\mathcal{C}$.

(dichotomie empruntée à Abramsky, cf. par ex. workshop *Structure meets Power*)

- Hillebrand & Kanellakis : $\mathcal{P} = \lambda$-c. simplement typé, $\mathcal{C}$ = langages rationnels
  - Bonne nouvelle : contrairement aux « polynômes étendus », un objet central d'un autre domaine de l'informatique, la *théorie des automates*
  - La définition sera rappelée bientôt
- *Complexité implicite* : $\mathcal{C}$ est une classe de complexité algorithmique (P, NP, ...)
  - un domaine actif de recherche depuis les années 1990 (cf. HDR de Péchoux)
  - Exemple historique (Girard) : $\mathcal{P}$ = logique linéaire allégée, $\mathcal{C} = \mathsf{P}$

« Automates implicites » : $\mathcal{C}$ provenant de la théorie des automates

Grandeur et misère de la complexité implicite

La complexité implicite est parvenue à caractériser de nombreuses classes de complexité ! Avec des langages de programmation variés...

Grandeur et misère de la complexité implicite

La complexité implicite est parvenue à caractériser de nombreuses classes de complexité ! Avec des langages de programmation variés… mais souvent 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

La complexité implicite est parvenue à caractériser de nombreuses classes de complexité ! Avec des langages de programmation variés… mais souvent 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*

Les automates finis surgissent « naturellement » en λ-calcul simplement typé !

$\longrightarrow$ on va essayer de préserver cet aspect…

Grandeur et misère de la complexité implicite

La complexité implicite est parvenue à caractériser de nombreuses classes de complexité ! Avec des langages de programmation variés… mais souvent 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*

Les automates finis surgissent « naturellement » en λ-calcul simplement typé !

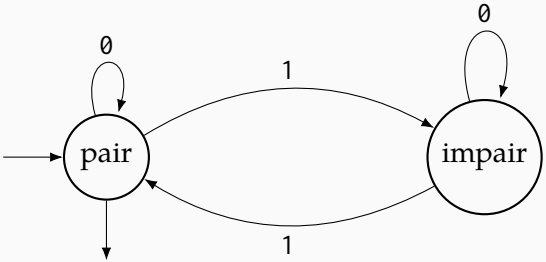
$\longrightarrow$ on va essayer de préserver cet aspect…

On commence par rappeler les *langages* rationnels, puis parler de notre nouvel objectif : leurs généralisations à des *fonctions* $\{\text{mots}\} \to \{\text{mots}\}$

Rappel : langages rationnels

La canonicité / robustesse de la classe des *langages rationnels* (*regular languages*) est attestée par ses nombreuses définitions équivalentes :

- *expressions rationnelles* : `0*(10*10*)*` = « que des `0` et `1` & nombre pair de `1` »
- *automates finis* (non-)déterministes : exemple ci-dessous



Rappel : langages rationnels

La canonicité / robustesse de la classe des *langages rationnels* (*regular languages*) est attestée par ses nombreuses définitions équivalentes :

- *expressions rationnelles* : `0*(10*10*)*` = « que des `0` et `1` & nombre pair de `1` »
- *automates finis* (non-)déterministes
- définition *algébrique* (proche des automates déterministes), ex : $M = \mathbb{Z}/2\mathbb{Z}$

Théorème (classique)

Un langage $L \subseteq \Sigma^*$ *est* rationnel $\iff$ *il existe un* morphisme $\varphi : \Sigma^* \to M$ *vers un* monoïde fini M *et une partie* $P \subseteq M$ *tels que* $L = \varphi^{-1}(P) = \{w \in \Sigma^* \mid \varphi(w) \in P\}$.

Rappel : langages rationnels

La canonicité / robustesse de la classe des *langages rationnels* (*regular languages*) est attestée par ses nombreuses définitions équivalentes :

- *expressions rationnelles* : `0*(10*10*)*` = « que des `0` et `1` & nombre pair de `1` »
- *automates finis* (non-)déterministes
- définition *algébrique* (proche des automates déterministes), ex : $M = \mathbb{Z}/2\mathbb{Z}$

Théorème (classique)

Un langage $L \subseteq \Sigma^*$ *est* rationnel $\iff$ *il existe un* morphisme $\varphi : \Sigma^* \to M$ *vers un* monoïde fini M *et une partie* $P \subseteq M$ *tels que* $L = \varphi^{-1}(P) = \{w \in \Sigma^* \mid \varphi(w) \in P\}$.

Quelle est la bonne généralisation à des fonctions $\Gamma^* \to \Sigma^*$ (*transductions*) ?

Classes habituelles de transductions

Quelle est la bonne généralisation des langages rationnels (en anglais : réguliers) à des *transductions* $\Gamma^* \to \Sigma^*$? Il y en a plusieurs !

$$\underbrace{\text{fonctions séquentielles}}_{\text{transducteurs finis } \textit{déterministes}} \subsetneq \underbrace{\text{fonctions rationnelles}}_{\text{transducteurs } \textit{non-déterministes}} \subsetneq \text{fonctions régulières}$$

Classes habituelles de transductions

Quelle est la bonne généralisation des langages rationnels (en anglais : réguliers) à des *transductions* $\Gamma^* \to \Sigma^*$? Il y en a plusieurs !

$$\underbrace{\text{fonctions séquentielles}}_{\text{transducteurs finis } \textit{déterministes}} \subsetneq \underbrace{\text{fonctions rationnelles}}_{\text{transducteurs } \textit{non-déterministes}} \subsetneq \text{fonctions régulières}$$

- ces classes sont *stables par composition*
- préservation : L rationnel $\implies f^{-1}(L)$ rationnel

Classes habituelles de transductions

Quelle est la bonne généralisation des langages rationnels (en anglais : réguliers)
à des *transductions* $\Gamma^* \to \Sigma^*$? Il y en a plusieurs !

$$\underbrace{\text{fonctions séquentielles}}_{\text{transducteurs finis } \textit{déterministes}} \subsetneq \underbrace{\text{fonctions rationnelles}}_{\text{transducteurs } \textit{non-déterministes}} \subsetneq \text{fonctions régulières}$$

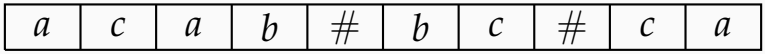
- ces classes sont *stables par composition*
- préservation : L rationnel $\implies f^{-1}(L)$ rationnel

On s'intéresse ici aux régulières ; plein de définitions (c'est canonique !),
par ex. *transducteurs à registres* (*streaming string transducers*)

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

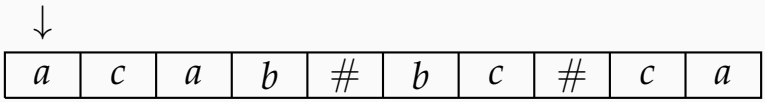


$X = \varepsilon \qquad Y = \varepsilon$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

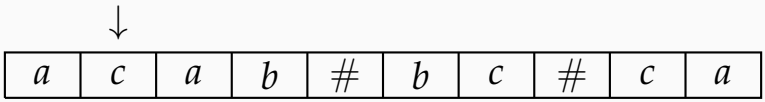


$X = a \qquad Y = \varepsilon$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

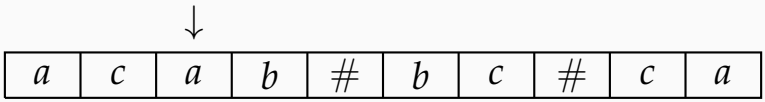


$$X = ca \qquad Y = \varepsilon$$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

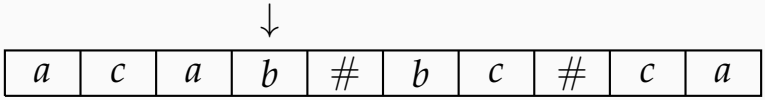


$$X = aca \qquad Y = \varepsilon$$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

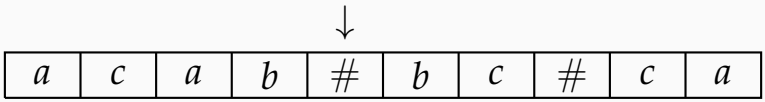


$$X = baca \qquad Y = \varepsilon$$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

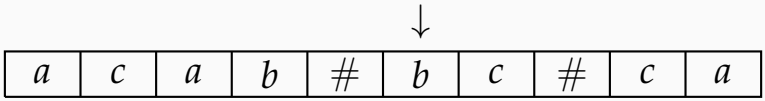


$$X = \varepsilon \qquad Y = baca\#$$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

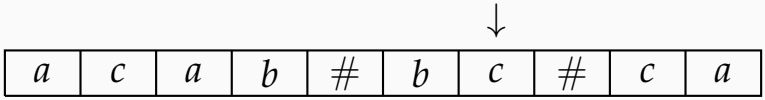


$X = b \qquad Y = baca\#$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

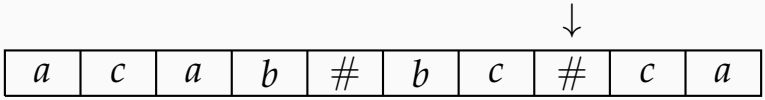


$X = cb \qquad Y = baca\#$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

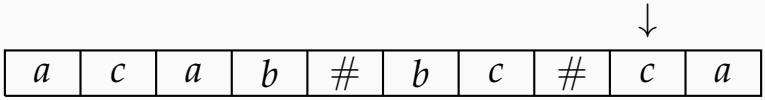


$$X = \varepsilon \qquad Y = baca\#cb\#$$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

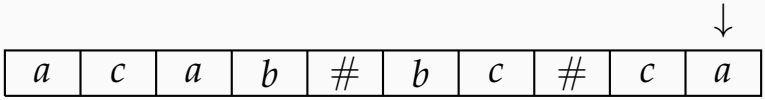


$$X = c \qquad Y = baca\#cb\#$$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

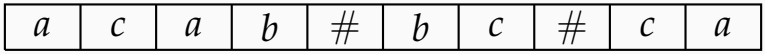


$$X = ac \qquad Y = baca\#cb\#$$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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

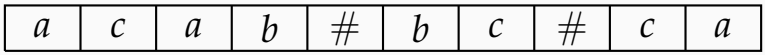


$X = ac \qquad Y = baca\#cb\# \qquad \texttt{mapReverse}(\ldots) = YX = baca\#cb\#ac$

Transducteurs à registres [Alur & Černý 2010]

Automate fini déterministe + *registres* contenant des mots. Exemple :

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



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

Fonctions régulières = calculées par transducteurs sans copie

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

chaque registre apparaît $\leqslant 1$ fois à droite d'un $:=$ dans une transition

Linéarité

Fonctions régulières = calculées par transducteurs *sans copie*

Restriction du « pouvoir de copie » : vieux thème de la théorie des automates

(par ex. Schützenberger, années 1960 : séries $\Sigma^* \to \mathbb{Z}$ rationnelles à coefficients $|w|^{O(1)}$)

Linéarité

Fonctions régulières = calculées par transducteurs *sans copie*

Restriction du « pouvoir de copie » : vieux thème de la théorie des automates

(par ex. Schützenberger, années 1960 : séries $\Sigma^* \to \mathbb{Z}$ rationnelles à coefficients $|w|^{O(1)}$)

Équivalent λ-calcul : typage linéaire (Girard 1987)

On va utiliser le « $\lambda\ell^{\oplus\&}$-calcul » = *Dual Intuitionistic Linear Logic*
+ connecteurs additifs $\oplus, \&$

(linéaire = *exactement* 1 fois ; les additifs permettent de simuler *au plus* 1 fois (affine))

Linéarité

Fonctions régulières = calculées par transducteurs *sans copie*

Restriction du « pouvoir de copie » : vieux thème de la théorie des automates

(par ex. Schützenberger, années 1960 : séries $\Sigma^* \to \mathbb{Z}$ rationnelles à coefficients $|w|^{O(1)}$)

Équivalent λ-calcul : typage linéaire (Girard 1987)

On va utiliser le « $\lambda\ell^{\oplus\&}$-calcul » = *Dual Intuitionistic Linear Logic*
+ connecteurs additifs $\oplus, \&$

(linéaire = *exactement* 1 fois ; les additifs permettent de simuler *au plus* 1 fois (affine))

$$A, B ::= o \mid \overbrace{\underbrace{A \multimap B}_{\text{fonctions linéaires}} \mid A \otimes B}^{\text{multiplicatifs}} \mid \overbrace{A \,\&\, B \mid A \oplus B}^{\text{additifs}} \mid \underbrace{A \to B}_{\text{fonctions non-linéaires}}$$

Premier résultat de transducteurs implicites

Codages de Church linéaires

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

Définition : un type du $\lambda\ell^{\oplus\&}$-calcul est *purement linéaire* s'il ne contient pas « $\to$ »

Théorème

$f : \Gamma^* \to \Sigma^*$ *est* régulière $\iff$ $\exists$ *un type* A purement linéaire *et* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *en* $\lambda\ell^{\oplus\&}$*-calcul tels que* $\forall w \in \Gamma^*$, $\overline{f(w)} =_\beta t\ \overline{w}$

Premier résultat de transducteurs implicites

Codages de Church linéaires

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

Définition : un type du $\lambda\ell^{\oplus\&}$-calcul est *purement linéaire* s'il ne contient pas « $\to$ »

Théorème

$f : \Gamma^* \to \Sigma^*$ *est* régulière $\iff$ $\exists$ *un type* A purement linéaire *et* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *en* $\lambda\ell^{\oplus\&}$*-calcul tels que* $\forall w \in \Gamma^*$, $\overline{f(w)} =_\beta t\ \overline{w}$

Et si on remplace $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ par $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$, quelle classe de fonctions ?

Premier résultat de transducteurs implicites

Codages de Church linéaires

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

Définition : un type du $\lambda\ell^{\oplus\&}$-calcul est *purement linéaire* s'il ne contient pas « $\to$ »

Théorème

$f : \Gamma^* \to \Sigma^*$ *est* régulière $\iff$ $\exists$ *un type* A purement linéaire *et* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *en* $\lambda\ell^{\oplus\&}$*-calcul tels que* $\forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$

Et si on remplace $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ par $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$, quelle classe de fonctions ?

- Mauvaise nouvelle : cette classe n'existait pas encore dans la littérature

Premier résultat de transducteurs implicites

Codages de Church linéaires

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

Définition : un type du $\lambda\ell^{\oplus\&}$-calcul est *purement linéaire* s'il ne contient pas « $\to$ »

Théorème

$f : \Gamma^* \to \Sigma^*$ *est* régulière $\iff$ $\exists$ *un type* A purement linéaire *et* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *en* $\lambda\ell^{\oplus\&}$*-calcul tels que* $\forall w \in \Gamma^*$, $\overline{f(w)} =_\beta t\ \overline{w}$

Et si on remplace $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ par $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$, quelle classe de fonctions ?

- Mauvaise nouvelle : cette classe n'existait pas encore dans la littérature
- Bonne nouvelle : elle est naturelle, surtout au vu des travaux récents sur les *fonctions polyrégulières* [Bojańczyk 2018]

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*

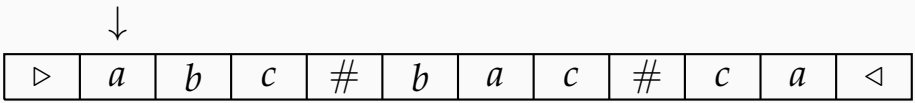
Sortie :

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



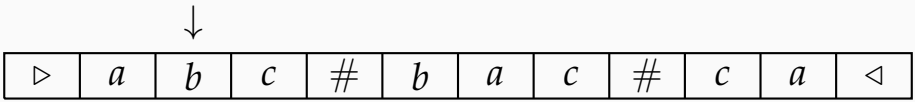
Sortie :

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



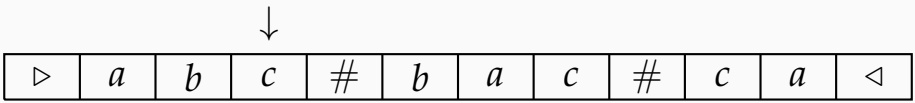
Sortie :

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



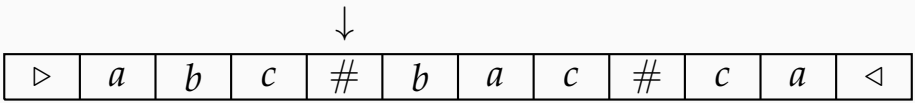
Sortie :

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



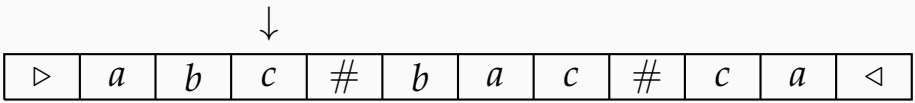
Sortie :

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



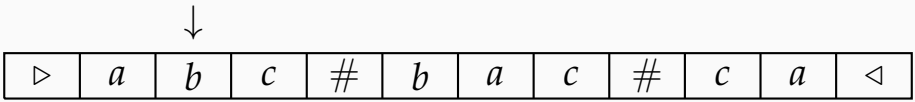
Sortie : c

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



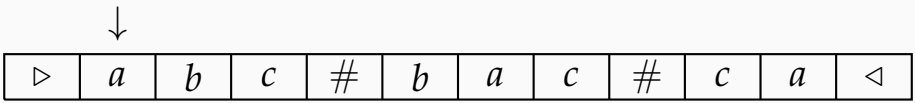
Sortie : cb

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



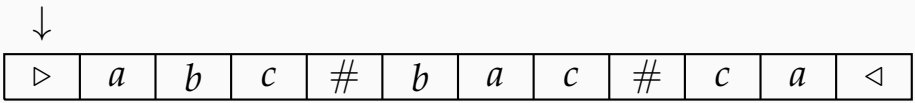
Sortie : cba

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



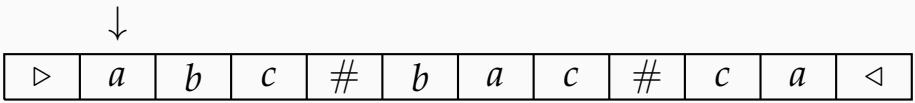
Sortie : *cba*

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



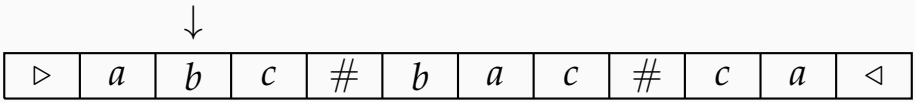
Sortie : cba

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



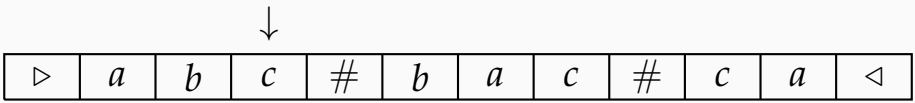
Sortie : *cba*

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



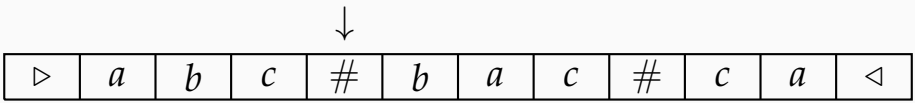
Sortie : *cba*

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



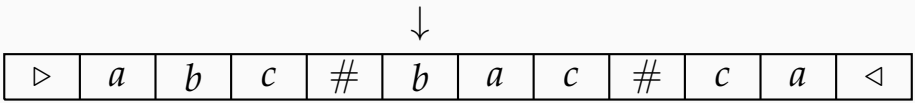
Sortie : $cba\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



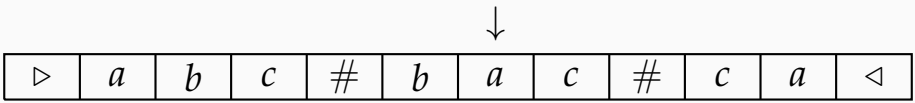
Sortie : $cba\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



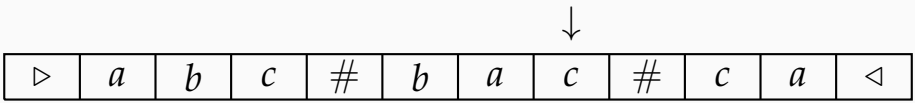
Sortie : $cba\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



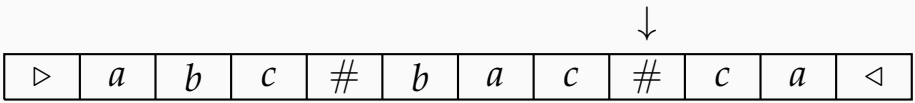
Sortie : $cba\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



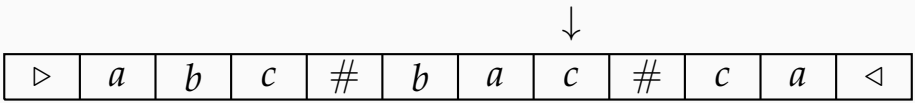
Sortie : $cba\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



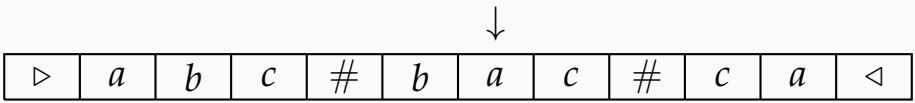
Sortie : $cba\#c$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



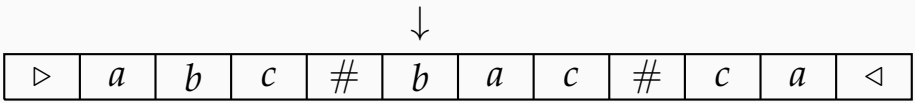
Sortie : $cba\#ca$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



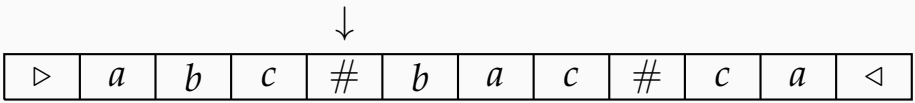
Sortie : $cba\#cab$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



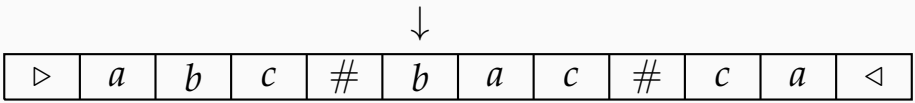
Sortie : $cba\#cab$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



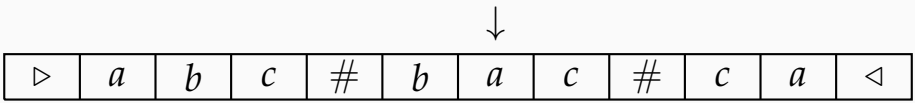
Sortie : $cba\#cab$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



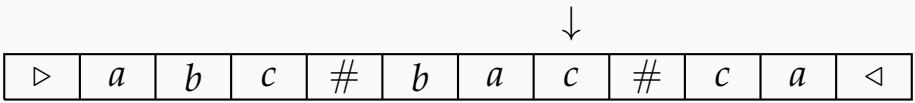
Sortie : $cba\#cab$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



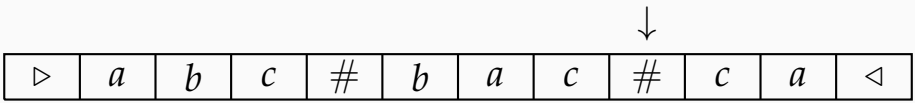
Sortie : $cba\#cab$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



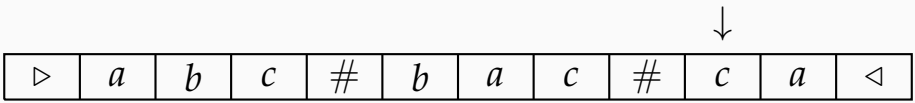
Sortie : $cba\#cab\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



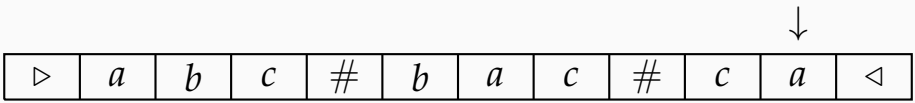
Sortie : $cba\#cab\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



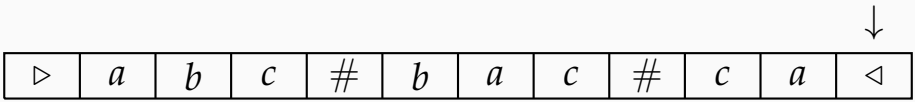
Sortie : $cba\#cab\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



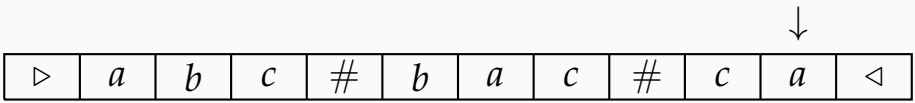
Sortie : $cba\#cab\#$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



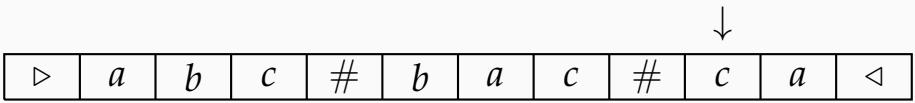
Sortie : $cba\#cab\#a$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



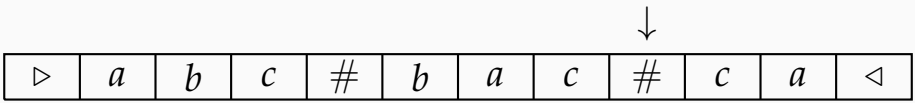
Sortie : $cba\#cab\#ac$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



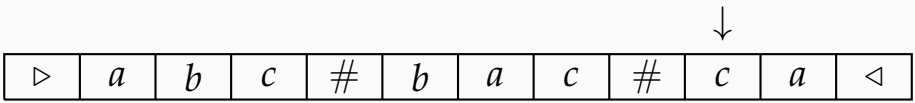
Sortie : $cba\#cab\#ac$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



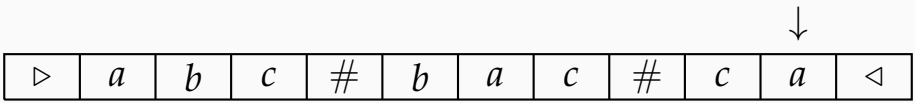
Sortie : $cba\#cab\#ac$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



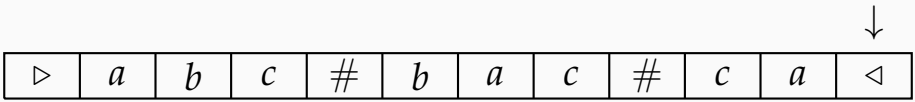
Sortie : $cba\#cab\#ac$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*



Sortie : $cba\#cab\#ac$

Un prérequis pour introduire les fonctions polyrégulières...

Transducteurs bidirectionnels (ou « boustrophédon » / EN : two-way)

Tête de lecture pouvant bouger à gauche ou à droite à chaque transition

$\longrightarrow$ autre définition équivalente des *fonctions régulières*

Sortie : $cba\#cab\#ac$

Remarque : ils sont reliés de près à la *géométrie de l'interaction* [Hines 2003], ce qui est utile pour les « automates implicites » (voir conclusion)

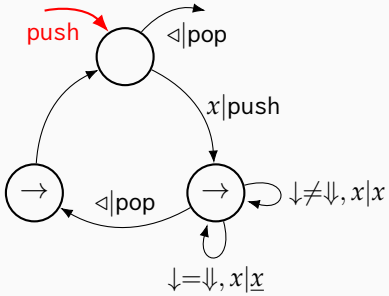
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

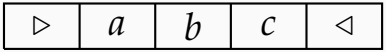
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie :

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

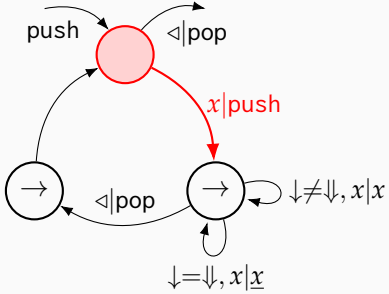
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

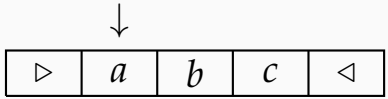
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie :

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

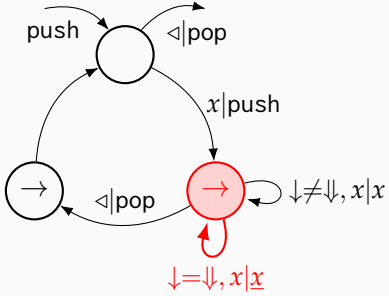
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

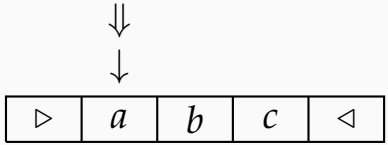
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie :

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

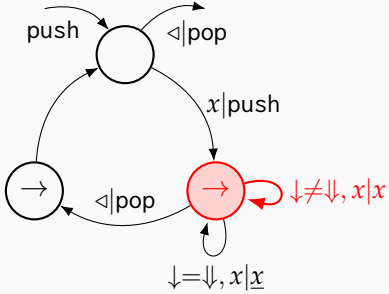
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

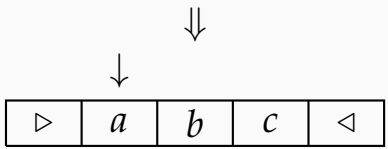
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie : $\underline{a}$

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

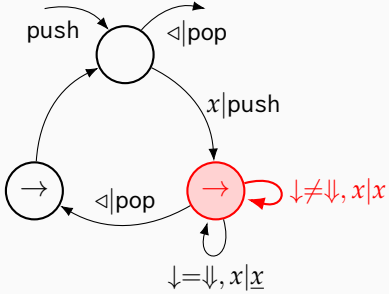
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

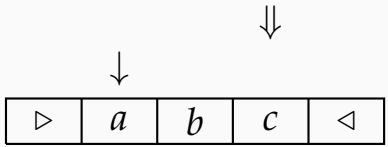
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie : $\underline{a}b$

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

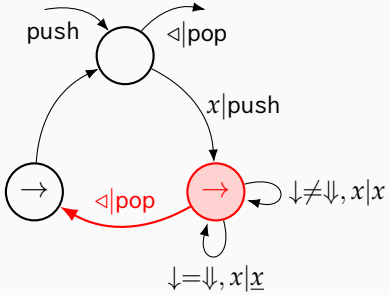
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

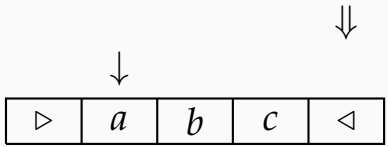
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie : $\underline{a}bc$

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

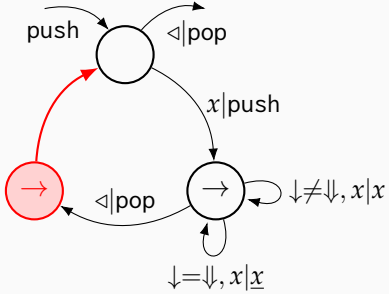
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

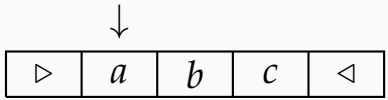
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie : $\underline{a}bc$

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

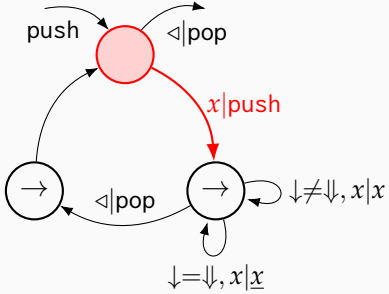
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

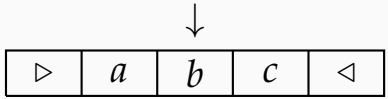
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie : $\underline{a}bc$

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

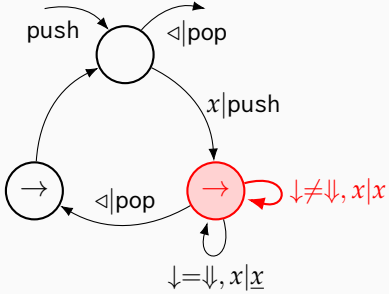
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

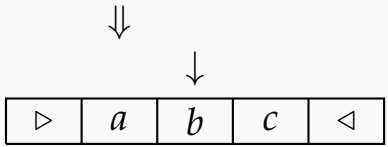
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie : $\underline{a}bc$

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

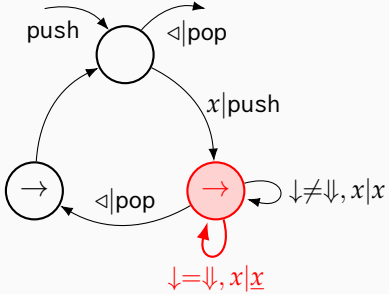
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

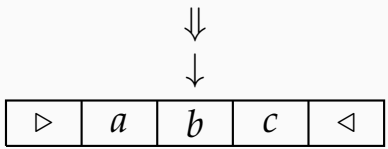
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





Sortie : $\underline{a}bca$

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

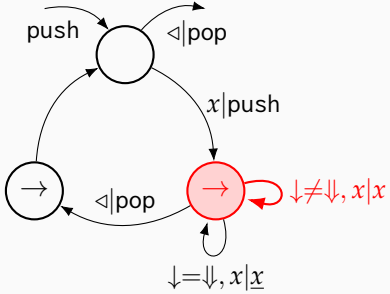
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

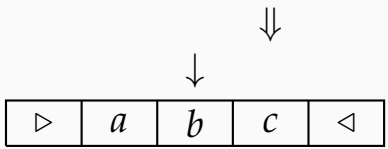
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

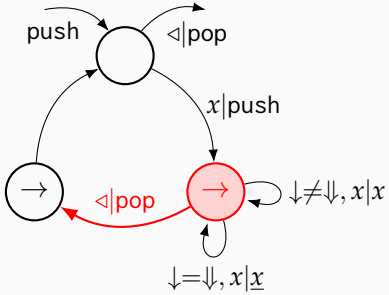
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

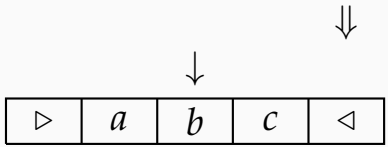
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

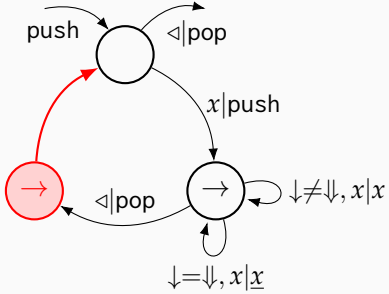
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

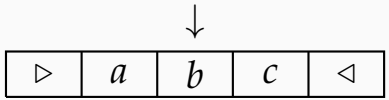
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

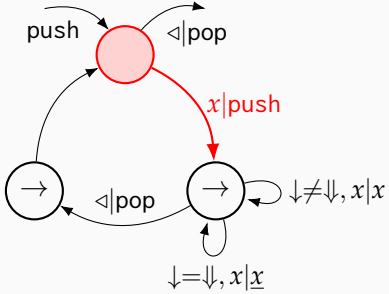
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

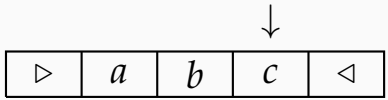
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

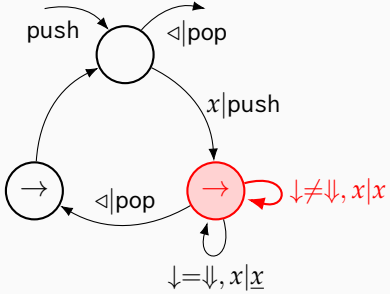
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

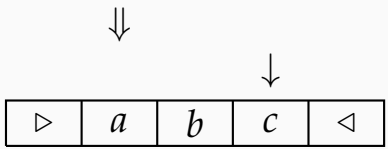
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

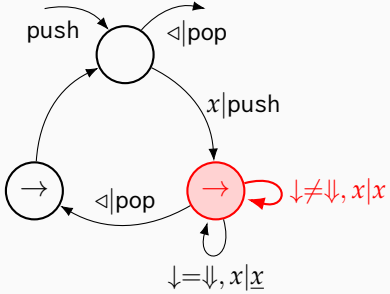
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

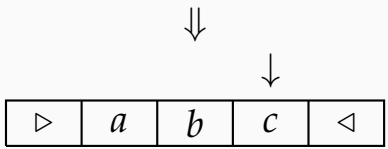
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

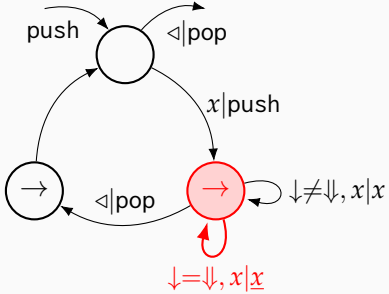
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

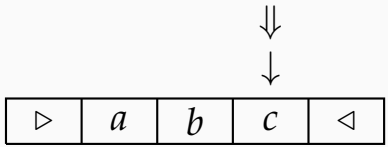
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

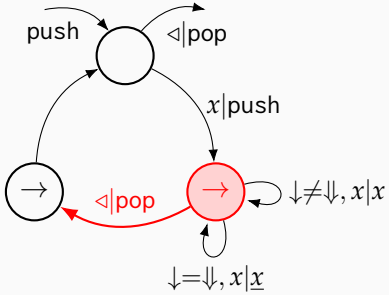
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

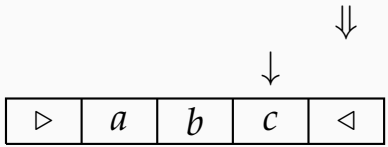
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

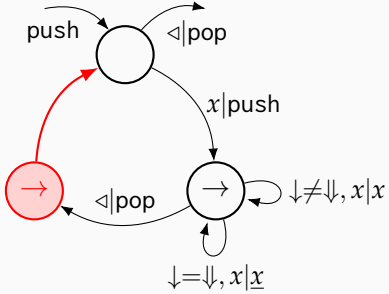
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

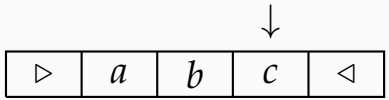
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

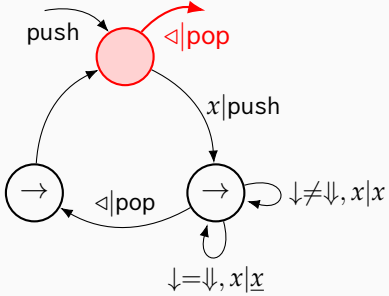
Transducteurs à jetons[1] (EN : pebble / PL : kamyki) [Milo, Suciu & Vianu 2000]

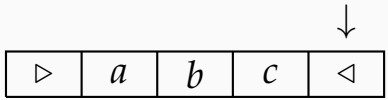
Fonctions polyrégulières = calculées par des transducteurs à k jetons

Ensemble fini d'états + *pile* de hauteur $\leqslant k$ de têtes bidirectionnelles («jetons»)

Les positions des têtes peuvent être *comparées*. Exemple :

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





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

[1] https://fr.wikipedia.org/wiki/Automate_à_jetons

Fonctions polyrégulières sans comparaison

Les *comparaisons* entre têtes de lecture (jetons) sont-elles vraiment nécessaires ?

Fonctions polyrégulières sans comparaison

Les *comparaisons* entre têtes de lecture (jetons) sont-elles vraiment nécessaires ?

- [N., Noûs & Pradic, ICALP'21] *oui*, par exemple pour calculer
$$f(a^n) = a\#aa\#\ldots\#a^n \text{ et } g(a^{n_1}\#\ldots\#a^{n_k}) = a^{n_1\times n_1}\#\ldots\#a^{n_k\times n_k}$$

Fonctions polyrégulières sans comparaison

Les *comparaisons* entre têtes de lecture (jetons) sont-elles vraiment nécessaires ?

- [N., Noûs & Pradic, ICALP'21] *oui*, par exemple pour calculer
$$f(a^n) = a\#aa\#\dots\#a^n \text{ et } g(a^{n_1}\#\dots\#a^{n_k}) = a^{n_1\times n_1}\#\dots\#a^{n_k\times n_k}$$
- Mais définit-on une classe de fonctions naturelle/canonique ainsi ?

Fonctions polyrégulières sans comparaison

Les *comparaisons* entre têtes de lecture (jetons) sont-elles vraiment nécessaires ?

- [N., Noûs & Pradic, ICALP'21] *oui*, par exemple pour calculer
  $f(a^n) = a\#aa\#\ldots\#a^n$ et $g(a^{n_1}\#\ldots\#a^{n_k}) = a^{n_1\times n_1}\#\ldots\#a^{n_k\times n_k}$
- Mais définit-on une classe de fonctions naturelle/canonique ainsi ? Oui !
  d'où leur étude « pour elles-mêmes », cf. exposé au LIS (Move) il y a 1 semaine

Théorème

$f : \Gamma^* \to \Sigma^*$ *est* polyrégulière sans comparaison

$\iff$

$\exists A$ purement linéaire *et* $t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ *tels que* $\forall w \in \Gamma^*$, $\overline{f(w)} =_\beta t\ \overline{w}$

Les connecteurs additifs sont-ils nécessaires ?

Les additifs $\oplus$/$\&$ servent :

- à représenter des conditions de « single use » plus subtiles que « sans copie » : surtout utile pour les transducteurs d'*arbres*

  (en fait, la distinction $\otimes$ vs $\&$ existe implicitement en théorie des automates)

Les connecteurs additifs sont-ils nécessaires ?

Les additifs $\oplus$/& servent :

- à représenter des conditions de « single use » plus subtiles que « sans copie » : surtout utile pour les transducteurs d'*arbres*

  (en fait, la distinction $\otimes$ vs & existe implicitement en théorie des automates)

- à coder les *états* d'un automate !

Les connecteurs additifs sont-ils nécessaires ?

Les additifs $\oplus$/& servent :

- à représenter des conditions de « single use » plus subtiles que « sans copie » : surtout utile pour les transducteurs d'*arbres*

  (en fait, la distinction $\otimes$ vs & existe implicitement en théorie des automates)

- à coder les *états* d'un automate !

$\longrightarrow$ perd-on de l'expressivité en enlevant les additifs ?

Les connecteurs additifs sont-ils nécessaires ?

Les additifs $\oplus$/& servent :

- à représenter des conditions de « single use » plus subtiles que « sans copie » : surtout utile pour les transducteurs d'*arbres*

  (en fait, la distinction $\otimes$ vs & existe implicitement en théorie des automates)
- à coder les *états* d'un automate !

$\longrightarrow$ perd-on de l'expressivité en enlevant les additifs ?

- sur les arbres, *oui*, on n'a pas tous les langages rationnels (voir conclusion)
- sur les mots, *non*, mais c'est subtil

Les connecteurs additifs sont-ils nécessaires ?

Les additifs $\oplus$/& servent :

- à représenter des conditions de « single use » plus subtiles que « sans copie » : surtout utile pour les transducteurs d'*arbres*

  (en fait, la distinction $\otimes$ vs & existe implicitement en théorie des automates)

- à coder les *états* d'un automate !

$\longrightarrow$ perd-on de l'expressivité en enlevant les additifs ?

- sur les arbres, *oui*, on n'a pas tous les langages rationnels (voir conclusion)
- sur les mots, *non*, mais c'est subtil

  on compense l'absence d'états par des théorèmes de *décomposition*

Décomposition des fonctions régulières

Une « structure » compositionnelle dans l'étude du « pouvoir » expressif :

Théorème (Bojańczyk et al. (voir par ex. [Bojańczyk & Stefański 2020]))

Toute fonction régulière peut être réalisée comme composition de :

- *fonctions* séquentielles, *voir diapo suivante*
- $\texttt{mapReverse}_\Sigma, \texttt{mapDuplicate}_\Sigma : (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ *for* $\# \notin \Sigma$

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

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

Décomposition des fonctions régulières

Une « structure » compositionnelle dans l'étude du « pouvoir » expressif :

Théorème (Bojańczyk et al. (voir par ex. [Bojańczyk & Stefański 2020]))

Toute fonction régulière peut être réalisée comme composition de :

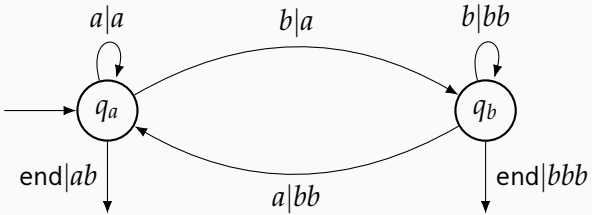
- *fonctions* séquentielles, *voir diapo suivante*
- $\texttt{mapReverse}_\Sigma, \texttt{mapDuplicate}_\Sigma : (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ *for* $\# \notin \Sigma$

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

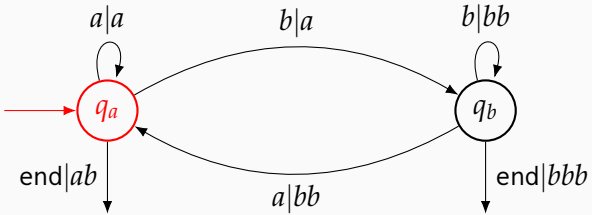
$\texttt{mapDuplicate}_\Sigma(w_1\# \ldots \#w_n) = w_1w_1\# \cdots \#w_nw_n$

- fonctions séquentielles $\to$ correspondent aux états finis
- $\texttt{mapReverse}, \texttt{mapDuplicate}$: calculées par transducteurs à registres sans états

Exemple de transducteur séquentiel

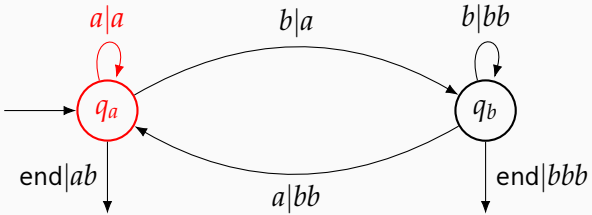


Exemple de transducteur séquentiel



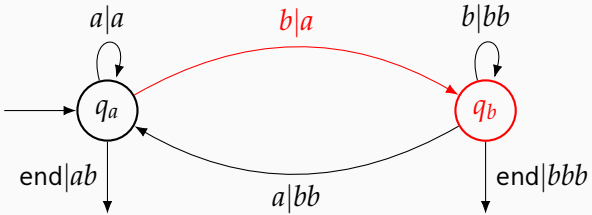
$abba \mapsto$

Exemple de transducteur séquentiel



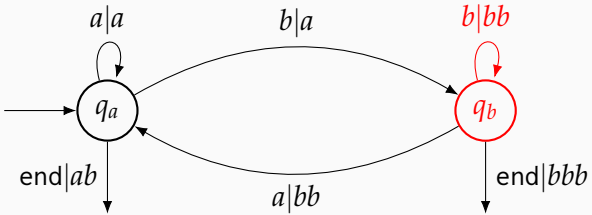
$abba \mapsto a$

Exemple de transducteur séquentiel



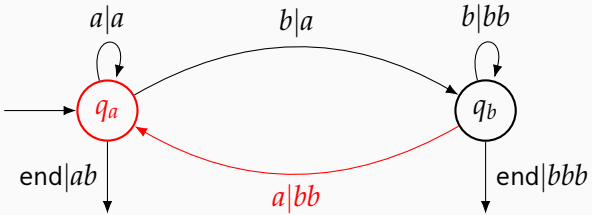
$abba \mapsto aa$

Exemple de transducteur séquentiel



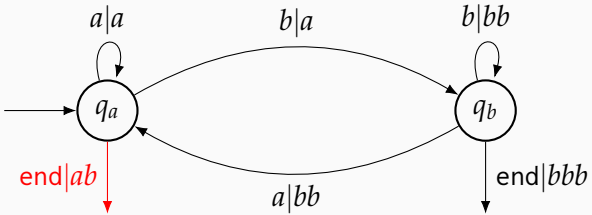
$$abba \mapsto aabb$$

Exemple de transducteur séquentiel



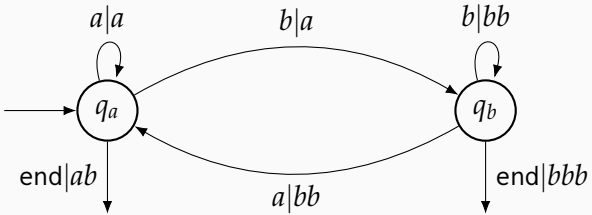
$$abba \mapsto aabbbb$$

Exemple de transducteur séquentiel



$$abba \mapsto aabbbbab$$

Exemple de transducteur séquentiel



$$abba \mapsto aabbbbab$$

Note : il existe une caractérisation implicite des fonctions séquentielles à base de *preuves cycliques* [DeYoung & Pfenning, APLAS'16]

Décomposition de Krohn–Rhodes

Théorème (Krohn & Rhodes 1965, légèrement reformulé)

Toute fonction séquentielle *peut être réalisée comme composition de transducteurs séq. :*

- *dont les transitions $Q \to Q$ sont bijectives (Q = ensemble d'états) ;*
- flip-flop *: à 2 états et sans* $(1 \mapsto 2; 2 \mapsto 1)$ *(par ex. celui de la diapo précédente).*

(*Fondamentalement, il s'agit d'une décomposition algébrique des monoïdes finis.*)

Il suffit donc de coder ces « briques de base » en λ-calcul *affine* par des termes de type $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$... puis ceux-ci se composent automatiquement !

Décomposition de Krohn–Rhodes

Théorème (Krohn & Rhodes 1965, légèrement reformulé)

Toute fonction séquentielle *peut être réalisée comme composition de transducteurs séq. :*

- *dont les transitions $Q \to Q$ sont bijectives (Q = ensemble d'états) ;*
- flip-flop *: à 2 états et sans* $(1 \mapsto 2; 2 \mapsto 1)$ *(par ex. celui de la diapo précédente).*

(*Fondamentalement, il s'agit d'une décomposition algébrique des monoïdes finis.*)

Il suffit donc de coder ces « briques de base » en λ-calcul *affine* par des termes de type $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$... puis ceux-ci se composent automatiquement !

Les *permutations* sur les états sont une primitive calculatoire importante

$\longrightarrow$ pouvoir calculatoire de la *règle d'échange* $f \rightsquigarrow \lambda x.\ \lambda y.\ f\ y\ x$

Décomposition de Krohn–Rhodes

Théorème (Krohn & Rhodes 1965, légèrement reformulé)

Toute fonction séquentielle *peut être réalisée comme composition de transducteurs séq. :*

- *dont les transitions $Q \to Q$ sont bijectives (Q = ensemble d'états) ;*
- flip-flop *: à 2 états et sans* $(1 \mapsto 2; 2 \mapsto 1)$ *(par ex. celui de la diapo précédente).*

(*Fondamentalement, il s'agit d'une décomposition algébrique des monoïdes finis.*)

Il suffit donc de coder ces « briques de base » en λ-calcul *affine* par des termes de type $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$... puis ceux-ci se composent automatiquement !

Les *permutations* sur les états sont une primitive calculatoire importante

$\longrightarrow$ pouvoir calculatoire de la *règle d'échange* $f \rightsquigarrow \lambda x.\ \lambda y.\ f\ y\ x$

Lien entre Krohn–Rhodes sans bijections et typage *non-commutatif* ?

Apériodicité

Théorème (Krohn & Rhodes 1965, cas apériodique)

Une fonction est séquentielle apériodique

$\iff$ *elle peut être réalisée comme composition de transducteurs séq. «flip-flop».*

Apériodicité

Théorème (Krohn & Rhodes 1965, cas apériodique)

Une fonction est séquentielle apériodique

$\iff$ *elle peut être réalisée comme composition de transducteurs séq. «flip-flop».*

Transducteur séquentiel apériodique = monoïde de transition apériodique

Définition

Un monoïde M est *apériodique* quand $\forall x \in M,\ \exists n \in \mathbb{N} : x^n = x^{n+1}$.

Apériodicité

Théorème (Krohn & Rhodes 1965, cas apériodique)

Une fonction est séquentielle apériodique
$\iff$ *elle peut être réalisée comme composition de transducteurs séq. «flip-flop».*

Transducteur séquentiel apériodique = monoïde de transition apériodique

Définition

Un monoïde M est *apériodique* quand $\forall x \in M,\ \exists n \in \mathbb{N} : x^n = x^{n+1}$.

Théorème (Schützenberger 1965)

Un langage $L \subseteq \Sigma^*$ *est* sans étoile $\iff$ $\exists$ *un* morphisme $\varphi : \Sigma^* \to M$ *vers un* monoïde fini et apériodique M *et* $P \subseteq M$ *tels que* $L = \varphi^{-1}(P)$.

(Rappel : sans «apériodique» $\rightsquigarrow$ langages rationnels)

Langages sans étoile

Un monoïde M est *apériodique* quand $\forall x \in M,\ \exists n \in \mathbb{N} : x^n = x^{n+1}$.

Théorème (Schützenberger 1965)

Un langage $L \subseteq \Sigma^*$ est sans étoile $\iff$ *$\exists$ un* morphisme $\varphi : \Sigma^* \to M$ *vers un* monoïde fini et apériodique M *et $P \subseteq M$ tels que $L = \varphi^{-1}(P)$.*

Expressions régulières sans étoile (de Kleene) – mais avec complémentation :

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

par ex. $(ab)^* = (b\varnothing^c \cup \varnothing^c a \cup \varnothing^c aa\varnothing^c \cup \varnothing^c bb\varnothing^c)^c$, mais $(aa)^*$ n'est pas sans étoile car son « monoïde minimal » $\mathbb{Z}/2\mathbb{Z}$ n'est pas apériodique

Automates implicites en logique non-commutative

Expressions régulières sans étoile : $L, L' ::= \varnothing \mid \{a\} \mid L \cdot L' \mid L \cup L' \mid L^{c}$

Théorème (N. & Pradic, ICALP 2020)

$L \subseteq \Sigma^*$ *est* sans étoile $\iff$ $\exists$ *un type* A purement affine *et* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Bool}$ *en « $\lambda\wp$-calcul » tels que* $L = \{w \in \Sigma^* \mid t\ \overline{w} =_{\beta\eta} \mathtt{true}\}$

Automates implicites en logique non-commutative

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

Théorème (N. & Pradic, ICALP 2020)

$L \subseteq \Sigma^*$ *est* sans étoile $\iff$ $\exists$ *un type* A purement affine *et* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Bool}$ *en « $\lambda\wp$-calcul » tels que* $L = \{w \in \Sigma^* \mid t\ \overline{w} =_{\beta\eta} \mathtt{true}\}$

$\lambda\wp$-calcul : variante affine de la *Intuitionistic Non-Commutative Linear Logic*

(thèse de Jeff Polakow, dir. Franck Pfenning)

Idée (remontant à Lambek 1958) : l'ordre des arguments compte

« une fonction $\lambda x.\ \lambda y.\ (\ldots)$ ne devrait utiliser x *après* y »

$\longrightarrow$ jugements de typage $\Phi \mid \Psi \vdash t : A$ pour un *ensemble* Φ (contexte non-linéaire) et une *liste ordonnée* Ψ (contexte linéaire)

Évaluation sémantique et théorie catégorique des automates

Comment passer des λ-termes typés aux automates ?

Évaluation sémantique et théorie catégorique des automates

Comment passer des λ-termes typés aux automates ?
Par la *sémantique dénotationnelle* : traduction λ-termes $\rightsquigarrow$ structures algébriques

- sémantique naïve du λ-calcul simplement typé dans les *ensembles finis*
  $\longrightarrow$ automates finis (théorème de Hillebrand & Kanellakis)

Évaluation sémantique et théorie catégorique des automates

Comment passer des λ-termes typés aux automates ?

Par la *sémantique dénotationnelle* : traduction λ-termes $\rightsquigarrow$ structures algébriques

- sémantique naïve du λ-calcul simplement typé dans les *ensembles finis*
  $\longrightarrow$ automates finis (théorème de Hillebrand & Kanellakis)

Approche pour nos résultats de « transducteurs implicites »

- Trouver une *catégorie monoïdale close* $\mathcal{C}$
  (donne une sémantique du fragment purement linéaire du $\lambda\ell^{\oplus\&}$-calcul)

Évaluation sémantique et théorie catégorique des automates

Comment passer des λ-termes typés aux automates ?

Par la *sémantique dénotationnelle* : traduction λ-termes $\rightsquigarrow$ structures algébriques

- sémantique naïve du λ-calcul simplement typé dans les *ensembles finis*
  $\longrightarrow$ automates finis (théorème de Hillebrand & Kanellakis)

Approche pour nos résultats de « transducteurs implicites »

- Trouver une *catégorie monoïdale close* $\mathcal{C}$
  (donne une sémantique du fragment purement linéaire du $\lambda\ell^{\oplus\&}$-calcul)
- telle que les *$\mathcal{C}$-automates* calculent les fonctions régulières
  (notion d'automate sur une catégorie ici : [Colcombet & Petrişan 2017])

Évaluation sémantique et théorie catégorique des automates

Comment passer des λ-termes typés aux automates ?

Par la *sémantique dénotationnelle* : traduction λ-termes $\rightsquigarrow$ structures algébriques

- sémantique naïve du λ-calcul simplement typé dans les *ensembles finis*
  $\longrightarrow$ automates finis (théorème de Hillebrand & Kanellakis)

Approche pour nos résultats de « transducteurs implicites »

- Trouver une *catégorie monoïdale close* $\mathcal{C}$
  (donne une sémantique du fragment purement linéaire du $\lambda\ell^{\oplus\&}$-calcul)
- telle que les *$\mathcal{C}$-automates* calculent les fonctions régulières
  (notion d'automate sur une catégorie ici : [Colcombet & Petrişan 2017])

Par ex., $\mathcal{C} = \mathsf{Int}(\mathsf{PFinSet}) \rightsquigarrow$ automates bidirectionnels [Hines 2003]
$\longrightarrow$ lien avec la géométrie de l'interaction, mais ne gère pas les additifs

Transducteurs à registres sans copie $\simeq$ $\mathcal{SR}_{\oplus}$-automates

- $\mathcal{SR}$ = catégorie des transitions sans copie entre ensembles finis de registres
- $(-)_{\oplus}$ = complétion libre par coproduits finis $\simeq$ ajoute des états finis

Transducteurs à registres sans copie $\simeq$ $\mathcal{SR}_\oplus$-automates

- $\mathcal{SR}$ = catégorie des transitions sans copie entre ensembles finis de registres
- $(-)_\oplus$ = complétion libre par coproduits finis $\simeq$ ajoute des états finis

- Souci : $\mathcal{SR}_\oplus$ n'est que « partiellement » monoïdale close…

Transducteurs à registres sans copie $\simeq$ $\mathcal{SR}_{\oplus}$-automates

- $\mathcal{SR}$ = catégorie des transitions sans copie entre ensembles finis de registres
- $(-)_{\oplus}$ = complétion libre par coproduits finis $\simeq$ ajoute des états finis

- Souci : $\mathcal{SR}_{\oplus}$ n'est que « partiellement » monoïdale close...
- Par contre, $(\mathcal{SR}_{\&})_{\oplus}$ est monoïdale close ($(-)_{\&}$ = complétion produit)

Catégories monoïdales closes et transducteurs à registres

Transducteurs à registres sans copie $\simeq \underline{\mathcal{SR}_\oplus}$-automates

- $\mathcal{SR}$ = catégorie des transitions sans copie entre ensembles finis de registres
- $(-)_\oplus$ = complétion libre par coproduits finis $\simeq$ ajoute des états finis

- Souci : $\mathcal{SR}_\oplus$ n'est que « partiellement » monoïdale close...
- Par contre, $(\mathcal{SR}_\&)_\oplus$ est monoïdale close ($(-)_\&$ = complétion produit)

Complétion $((-)_\&)_\oplus \simeq$ ajout d'états avec <u>non-déterminisme</u>

- les trans. à registres sans copie sont déterminisables [Alur & Deshmukh 2011]

Catégories monoïdales closes et transducteurs à registres

Transducteurs à registres sans copie $\simeq$ $\mathcal{SR}_\oplus$-automates

- $\mathcal{SR}$ = catégorie des transitions sans copie entre ensembles finis de registres
- $(-)_\oplus$ = complétion libre par coproduits finis $\simeq$ ajoute des états finis

- Souci : $\mathcal{SR}_\oplus$ n'est que « partiellement » monoïdale close...
- Par contre, $(\mathcal{SR}_\&)_\oplus$ est monoïdale close ($(-)_\&$ = complétion produit)

Complétion $((-)_\&)_\oplus \simeq$ ajout d'états avec non-déterminisme

- les trans. à registres sans copie sont déterminisables [Alur & Deshmukh 2011]
- **contribution :** déterminisation catégorique utilisant l'existence de certains « espaces de fonctions » $A \multimap B$ = clôture monoïdale « partielle » !

Structure monoïdale close de la catégorie $(\mathcal{SR}_{\&})_{\oplus}$

Pour $A, B \in \mathsf{Obj}(\mathcal{SR})$: décomposition des morphismes (technique classique)

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

sans copie $\implies$ nombre *fini* de formes possibles $\implies$ $A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$

Structure monoïdale close de la catégorie $(\mathcal{SR}_{\&})_{\oplus}$

Pour $A, B \in \mathsf{Obj}(\mathcal{SR})$: décomposition des morphismes (technique classique)

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

$$\textit{sans copie} \implies \text{nombre } \textit{fini} \text{ de formes possibles} \implies A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$$

Théorème (autre application de cette technique [N., Noûs, Pradic ICALP'21])

Les fonctions polyrégulières sans comparaison sont stables par composition.

Structure monoïdale close de la catégorie $(\mathcal{SR}_{\&})_{\oplus}$

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

Théorème (construction « à la Dialectica » inspirée de [Gödel, de Paiva, Hofstra])

Soit $\mathcal{C}$ une catégorie monoïdale symétrique. Si un espace de fonctions $A \multimap B$ existe dans $\mathcal{C}_{\oplus}$ pour tous $A, B \in \mathsf{Obj}(\mathcal{C})$, alors $(\mathcal{C}_{\&})_{\oplus}$ est monoïdale close.

Pour $A, B \in \mathsf{Obj}(\mathcal{SR})$: décomposition des morphismes (technique classique)

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \rightsquigarrow \quad \text{forme} \begin{cases} X := Z_1XZ_2Y \\ Y := Z_3 \end{cases} \quad + \quad \text{étiquettes } Z_1 = ab, \ldots$$

sans copie $\implies$ nombre *fini* de formes possibles $\implies$ $A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$

Théorème (autre application de cette technique [N., Noûs, Pradic ICALP'21])

Les fonctions polyrégulières sans comparaison sont stables par composition.

Conclusion

- caractérisations de classes de transductions dans un λ-calcul linéaire
  + langages sans étoile avec un typage non-commutatif
- lien entre sémantique dénotationnelle et théorie catégorique des automates
- découverte de la classe des fonctions polyrégulières sans comparaison

Conclusion

- caractérisations de classes de transductions dans un λ-calcul linéaire
  + langages sans étoile avec un typage non-commutatif
- lien entre sémantique dénotationnelle et théorie catégorique des automates
- découverte de la classe des fonctions polyrégulières sans comparaison

Conclusion

- caractérisations de classes de transductions dans un λ-calcul linéaire
  + langages sans étoile avec un typage non-commutatif
- lien entre sémantique dénotationnelle et théorie catégorique des automates
- découverte de la classe des fonctions polyrégulières sans comparaison

Plein de perspectives : la brèche est ouverte, reste à s'y engouffrer !

- vieux problème ouvert : $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ en λ-calcul simplement typé
  $\longrightarrow$ liens avec le model-checking d'ordre supérieur ?

Conclusion

- caractérisations de classes de transductions dans un λ-calcul linéaire
  + langages sans étoile avec un typage non-commutatif
- lien entre sémantique dénotationnelle et théorie catégorique des automates
- découverte de la classe des fonctions polyrégulières sans comparaison

Plein de perspectives : la brèche est ouverte, reste à s'y engouffrer !

- vieux problème ouvert : $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ en λ-calcul simplement typé
  $\longrightarrow$ liens avec le model-checking d'ordre supérieur ?
- λ-c. affine sans additifs $\underbrace{\subseteq}_{\text{géométrie de l'interaction}}$ automates cheminants $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ lang. rat. d'*arbres*

Conclusion

- caractérisations de classes de transductions dans un λ-calcul linéaire
  + langages sans étoile avec un typage non-commutatif
- lien entre sémantique dénotationnelle et théorie catégorique des automates
- découverte de la classe des fonctions polyrégulières sans comparaison

Plein de perspectives : la brèche est ouverte, reste à s'y engouffrer !

- vieux problème ouvert : $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ en λ-calcul simplement typé
  $\longrightarrow$ liens avec le model-checking d'ordre supérieur ?
- λ-c. affine sans additifs $\underbrace{\subseteq}_{\text{géométrie de l'interaction}}$ automates cheminants $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ lang. rat. d'*arbres*
- atomes (ensembles nominaux), λ-calcul parcimonieux, polymorphisme, ...

Merci pour votre attention !

- caractérisations de classes de transductions dans un λ-calcul linéaire
  + langages sans étoile avec un typage non-commutatif
- lien entre sémantique dénotationnelle et théorie catégorique des automates
- découverte de la classe des fonctions polyrégulières sans comparaison

Plein de perspectives : la brèche est ouverte, reste à s'y engouffrer !

- vieux problème ouvert : $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ en λ-calcul simplement typé
  $\longrightarrow$ liens avec le model-checking d'ordre supérieur ?
- λ-c. affine sans additifs $\underbrace{\subseteq}_{\text{géométrie de l'interaction}}$ automates cheminants $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ lang. rat. d'*arbres*
- atomes (ensembles nominaux), λ-calcul parcimonieux, polymorphisme, ...