Implicit automata in typed λ-calculi

Nguyễn Lê Thành Dũng (a.k.a. Tito) — nltd@nguyentito.eu
Laboratoire d'informatique de Paris Nord, Villetaneuse, France
joint work with Pierre Pradic (University of Oxford)

Inria Saclay / LIX, Partout seminar, May 4th, 2020

The idea

Suppose the programs of type T in a programming language $\mathcal{P}$ all compute *languages*, something like $T = \texttt{String} \to \texttt{Bool}$.
(Or *functions* $\texttt{String} \to \texttt{String}$.)

What class of languages? Depends on $\mathcal{P}$ and T.
Turing-completeness: all computable languages.

The idea

Suppose the programs of type T in a programming language $\mathcal{P}$ all compute *languages*, something like $T = \texttt{String} \to \texttt{Bool}$.
(Or *functions* $\texttt{String} \to \texttt{String}$.)

What class of languages? Depends on $\mathcal{P}$ and T.
Turing-completeness: all computable languages.
Many theoretical PLs *not* Turing-complete.
(type systems $\to$ guaranteed termination w/ time bounds)

The idea

Suppose the programs of type T in a programming language $\mathcal{P}$ all compute *languages*, something like $T = \texttt{String} \to \texttt{Bool}$.
(Or *functions* $\texttt{String} \to \texttt{String}$.)

What class of languages? Depends on $\mathcal{P}$ and T.
Turing-completeness: all computable languages.
Many theoretical PLs *not* Turing-complete.
(type systems $\to$ guaranteed termination w/ time bounds)

Implicit computational complexity: machine-free characterizations of complexity classes using PLs

The idea

Suppose the programs of type T in a programming language $\mathcal{P}$ all compute *languages*, something like $T = \texttt{String} \to \texttt{Bool}$.
(Or *functions* $\texttt{String} \to \texttt{String}$.)

What class of languages? Depends on $\mathcal{P}$ and T.
Turing-completeness: all computable languages.
Many theoretical PLs *not* Turing-complete.
(type systems $\to$ guaranteed termination w/ time bounds)

Implicit computational complexity: machine-free characterizations of complexity classes using PLs

My research project w/ Pierre Pradic:
The same thing for **automata theory** instead of complexity

Previous work

Template for a theorem

The languages/functions computed by programs of type T in $\mathcal{P}$ are the same as those computed by the class $\mathcal{A}$ of automata.

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages

[1] Hillebrand & Kanellakis 1996

Previous work

Template for a theorem

The languages/functions computed by programs of type T in $\mathcal{P}$ are the same as those computed by the class $\mathcal{A}$ of automata.

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages

[1] Hillebrand & Kanellakis 1996

...

Previous work

Template for a theorem

The languages/functions computed by programs of type T in $\mathcal{P}$ are the same as those computed by the class $\mathcal{A}$ of automata.

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages
[2]	cyclic proofs ("$\texttt{Str} \to \texttt{Str}$")	sequential transducers

[1] Hillebrand & Kanellakis 1996

...

[2] DeYoung & Pfenning 2016

Previous work

Template for a theorem

The languages/functions computed by programs of type T in $\mathcal{P}$ are the same as those computed by the class $\mathcal{A}$ of automata.

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages
[2]	cyclic proofs ("`Str` $\to$ `Str`")	sequential transducers
[3]	cyclic proofs ("`Str` $\to$ `Bool`")	regular languages
		(add non-linearity $\to$ logspace)

[1] Hillebrand & Kanellakis 1996

...

[2] DeYoung & Pfenning 2016

[3] Kuperberg, Pinault & Pous 2019 (independently of [2])

Previous work

Template for a theorem

The languages/functions computed by programs of type T in $\mathcal{P}$ are the same as those computed by the class $\mathcal{A}$ of automata.

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages
[2]	cyclic proofs ("`Str` $\to$ `Str`")	sequential transducers
[3]	cyclic proofs ("`Str` $\to$ `Bool`")	regular languages
		(add non-linearity $\to$ logspace)

[1] Hillebrand & Kanellakis 1996 $\to$ **let's start with this**

...

[2] DeYoung & Pfenning 2016

[3] Kuperberg, Pinault & Pous 2019 (independently of [2])

Defining languages in the simply typed λ-calculus

We use *Church encodings* [Böhm & Berarducci 1985]:

Binary strings: think of `fold_right` on a list of characters

$$\texttt{011} \rightsquigarrow (f_0, f_1, x) \mapsto f_0 \circ f_1 \circ f_1(x)$$

$$\overline{\texttt{011}} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x))$$

Defining languages in the simply typed λ-calculus

We use *Church encodings* [Böhm & Berarducci 1985]:

Binary strings: think of `fold_right` on a list of characters

$$\texttt{011} \rightsquigarrow (f_0, f_1, x) \mapsto f_0 \circ f_1 \circ f_1(x)$$

$$\overline{\texttt{011}} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \quad \text{for any type } A$$

$$\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] = (A \to A) \to (A \to A) \to A \to A$$

Defining languages in the simply typed λ-calculus

We use *Church encodings* [Böhm & Berarducci 1985]:

Binary strings: think of `fold_right` on a list of characters

$$\texttt{011} \rightsquigarrow (f_0, f_1, x) \mapsto f_0 \circ f_1 \circ f_1(x)$$

$$\overline{\texttt{011}} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \quad \text{for any type } A$$

$$\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] = (A \to A) \to (A \to A) \to A \to A$$

Booleans: think of `if` b `then` x `else` y

$\texttt{true} : \lambda x.\ \lambda y.\ x \quad \texttt{false} : \lambda x.\ \lambda y.\ y \quad \mathsf{Bool}[A] = A \to A \to A$

Defining languages in the simply typed λ-calculus

We use *Church encodings* [Böhm & Berarducci 1985]:

Binary strings: think of `fold_right` on a list of characters

$$\texttt{011} \rightsquigarrow (f_0, f_1, x) \mapsto f_0 \circ f_1 \circ f_1(x)$$

$$\overline{\texttt{011}} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \quad \text{for any type } A$$

$$\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] = (A \to A) \to (A \to A) \to A \to A$$

Booleans: think of `if` b `then` x `else` y

$\texttt{true} : \lambda x.\ \lambda y.\ x \quad \texttt{false} : \lambda x.\ \lambda y.\ y \quad \mathsf{Bool}[A] = A \to A \to A$

Languages: $\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \to \mathsf{Bool}[o]$ (A = any type, Bool = base type)

Defining languages in the simply typed λ-calculus

We use *Church encodings* [Böhm & Berarducci 1985]:

Binary strings: think of `fold_right` on a list of characters

$$\texttt{011} \rightsquigarrow (f_0, f_1, x) \mapsto f_0 \circ f_1 \circ f_1(x)$$

$$\overline{\texttt{011}} = \lambda f_0.\ \lambda f_1.\ \lambda x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \quad \text{for any type } A$$

$$\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] = (A \to A) \to (A \to A) \to A \to A$$

Booleans: think of `if` b `then` x `else` y

$\texttt{true} : \lambda x.\ \lambda y.\ x \quad \texttt{false} : \lambda x.\ \lambda y.\ y \quad \mathsf{Bool}[A] = A \to A \to A$

Languages: $\mathsf{Str}_{\{\texttt{0},\texttt{1}\}}[A] \to \mathsf{Bool}[o]$ (A = any type, Bool = base type)

Theorem (Hillebrand & Kanellakis 1996)

This can define all the regular languages*, and only those.*

Regular languages

Many equivalent definitions:

- *regular expressions*: `.*(a.*a.*)*` = "even number of 'a's"
- *finite automata* (DFA/NFA): too lazy to draw one of these

Regular languages

Many equivalent definitions:

- *regular expressions*: `.*(a.*a.*)*` = "even number of 'a's"
- *finite automata* (DFA/NFA): too lazy to draw one of these
- *algebraic* definition below

Theorem (classical)

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

Σ: finite alphabet, Σ^*: words over Σ

monoid structure: for $v, w \in \Sigma^*$, $v \cdot w$ = concatenation

morphism: for $w \in \Sigma^*$ with n letters, $\varphi(w) = \varphi(w[0]) \dots \varphi(w[n])$

Regular languages

Many equivalent definitions:

- *regular expressions*: `.*(a.*a.*)*` = "even number of 'a's"
- *finite automata* (DFA/NFA): too lazy to draw one of these
- *algebraic* definition below
- simply typed λ-calculus + Church encodings!

Theorem (classical)

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

Σ: finite alphabet, Σ^*: words over Σ

monoid structure: for $v, w \in \Sigma^*$, $v \cdot w$ = concatenation

morphism: for $w \in \Sigma^*$ with n letters, $\varphi(w) = \varphi(w[0]) \dots \varphi(w[n])$

λ-definable languages are regular

Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A & any simply typed λ-term $t : \mathsf{Str}_{\{0,1\}}[A] \to \mathsf{Bool}[o]$, the language $\{w \in \{0,1\}^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is regular.

Proof sketch: by semantic evaluation.

Use the standard semantics in *finite sets*: $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$, so that $[\![\mathsf{Str}[A]]\!]$ is finite.

$[\![\overline{w}]\!]$ determines whether $[\![t\,\overline{w}]\!] = [\![t]\!]\,([\![\overline{w}]\!]) = [\![\mathtt{true}]\!]$. When $\mathrm{Card}([\![o]\!]) \geq 2$, $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$, so this means $t\,\overline{w} =_\beta \mathtt{true}$.

□

λ-definable languages are regular

Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A & any simply typed λ-term $t : \mathsf{Str}_{\{0,1\}}[A] \to \mathsf{Bool}[o]$, the language $\{w \in \{0,1\}^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is regular.

Proof sketch: by semantic evaluation.

Use the standard semantics in *finite sets*: $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$, so that $[\![\mathsf{Str}[A]]\!]$ is finite.

$[\![\overline{w}]\!]$ determines whether $[\![t\,\overline{w}]\!] = [\![t]\!]\,([\![\overline{w}]\!]) = [\![\mathtt{true}]\!]$. When $\mathrm{Card}([\![o]\!]) \geq 2$, $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$, so this means $t\,\overline{w} =_\beta \mathtt{true}$.

$[\![\mathsf{Str}[A]]\!]$ has a monoid structure $([\![\mathsf{concat}]\!], [\![\overline{\varepsilon}]\!])$ such that $w \in \{0,1\}^* \to [\![\overline{w}]\!] \in [\![\mathsf{Str}[A]]\!]$ is a monoid morphism. □

λ-definable languages are regular

Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A & any simply typed λ-term $t : \mathsf{Str}_{\{0,1\}}[A] \to \mathsf{Bool}[o]$, the language $\{w \in \{0,1\}^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is regular.

Proof sketch: by semantic evaluation.

Use the standard semantics in *finite sets*: $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$, so that $[\![\mathsf{Str}[A]]\!]$ is finite.

$[\![\overline{w}]\!]$ determines whether $[\![t\,\overline{w}]\!] = [\![t]\!]\,([\![\overline{w}]\!]) = [\![\mathtt{true}]\!]$. When $\mathrm{Card}([\![o]\!]) \geq 2$, $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$, so this means $t\,\overline{w} =_\beta \mathtt{true}$.

$[\![\mathsf{Str}[A]]\!]$ has a monoid structure $([\![\mathrm{concat}]\!], [\![\overline{\varepsilon}]\!])$ such that $w \in \{0,1\}^* \to [\![\overline{w}]\!] \in [\![\mathsf{Str}[A]]\!]$ is a monoid morphism. □

Similar ideas in higher-order model checking, e.g. Grellois & Melliès

Our work

Template for a theorem

The languages/functions computed by programs of type T in $\mathcal{P}$ are the same as those computed by the class $\mathcal{A}$ of automata.

$\mathcal{P}$	$\mathcal{A}$
non-commutative λ-calculus	star-free languages, aperiodic reg. fun.
(commutative) affine λ-calculus	regular functions

Star-free languages: ICALP 2020 paper.

Other items in preparation.

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star*

e.g. for $\Sigma = \{a, b\}$, $(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c$

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star*

e.g. for $\Sigma = \{a, b\}$, $(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c$

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

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star*

e.g. for $\Sigma = \{a, b\}$, $(ab)^* = (b\varnothing^c|\varnothing^c a|\varnothing^c aa\varnothing^c|\varnothing^c bb\varnothing^c)^c$

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

Theorem (Schützenberger 1965)

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

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star*

e.g. for $\Sigma = \{a, b\}$, $(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa\varnothing^c | \varnothing^c bb\varnothing^c)^c$

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

Theorem (Schützenberger 1965)

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

Definition

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

Morally, $(aa)^*$ involves the group $\mathbb{Z}/(2)$: not aperiodic

From aperiodicity to non-commutativity

How to enforce aperiodicity in a typed λ-calculus?
Embedding of $\mathbb{Z}/(2)$ via $\mathtt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\mathtt{not} \circ \mathtt{not} =_\beta \mathtt{id}$)

From aperiodicity to non-commutativity

How to enforce aperiodicity in a typed λ-calculus?
Embedding of $\mathbb{Z}/(2)$ via $\mathtt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\mathtt{not} \circ \mathtt{not} =_\beta \mathtt{id}$)

$$\mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

morally, $\mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \rightsquigarrow \mathtt{if}\ \mathrm{not}(b)\ \mathtt{then}\ y\ \mathtt{else}\ x$
the not function *exchanges* two of its arguments

From aperiodicity to non-commutativity

How to enforce aperiodicity in a typed λ-calculus?
Embedding of $\mathbb{Z}/(2)$ via $\mathtt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\mathtt{not} \circ \mathtt{not} =_\beta \mathtt{id}$)

$$\mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

morally, $\mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \rightsquigarrow \mathtt{if}\ \mathrm{not}(b)\ \mathtt{then}\ y\ \mathtt{else}\ x$
the not function *exchanges* two of its arguments

Idea: make the order of arguments matter

"a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should first use b, then x, then y"
$\longrightarrow$ non-commutative type system

(Lambek 1958: non-commutative λ-calculus for linguistics)

Non-commutative types and linear logic

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

Non-commutative types and linear logic

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

$$\lambda f.\ \lambda x.\ \lambda y.\ (\lambda a.\ f\ a\ a)\ (x\ y) =_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

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

Non-commutative types and linear logic

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

$$\lambda f.\ \lambda x.\ \lambda y.\ (\lambda a.\ f\ a\ a)\ (x\ y) =_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

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

Fix: prohibit duplication, as in linear logic

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

Non-commutative types and linear logic

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

$$\lambda f.\ \lambda x.\ \lambda y.\ (\lambda a.\ f\ a\ a)\ (x\ y) =_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$$

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

Fix: prohibit duplication, as in linear logic

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

(If "*exactly once*": non-commutative *linear* λ-calculus;
correspondence with *planar combinatorial maps* (N. Zeilberger))

Finally, our theorem

Our type system: a base type o + *two* function arrows:

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

$$\mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A] = (A \multimap A) \to (A \multimap A) \to (A \multimap A)$$

$$\overline{\mathtt{011}} = \lambda^{\rightarrow} f_0.\ \lambda^{\rightarrow} f_1.\ \lambda^{\circ} x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A]$$

Finally, our theorem

Our type system: a base type o + *two* function arrows:

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

$$\mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A] = (A \multimap A) \to (A \multimap A) \to (A \multimap A)$$

$$\overline{\mathtt{011}} = \lambda^{\rightarrow} f_0.\ \lambda^{\rightarrow} f_1.\ \lambda^{\circ} x.\ f_0\ (f_1\ (f_1\ x)) : \mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A]$$

Theorem (N. & Pradic, ICALP 2020)

This typed λ-calculus can define all star-free languages*, and only those, with terms of type* $\mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A] \multimap \mathsf{Bool}[o]$ *where A is* purely affine *i.e. does not contain any unrestricted function arrow.*

(*A may vary depending on the language, as in Hillebrand & Kanellakis.*)

With commutative affine types, you'd get regular languages.

A non-commutative affine type system

Typing judgments $\Gamma \mid \Delta \vdash t : A$ for a *set* Γ and an **ordered list** Δ

$$\frac{}{\Gamma \uplus \{x : A\} \mid \varnothing \vdash x : A} \qquad \frac{}{\Gamma \mid x : A \vdash x : A}$$

$$\frac{\Gamma \mid \Delta \vdash t : A \to B \qquad \Gamma \mid \varnothing \vdash u : A}{\Gamma \mid \Delta \vdash t\,u : B} \qquad \frac{\Gamma \uplus \{x : A\} \mid \Delta \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\to} x.\, t : A \to B}$$

$$\frac{\Gamma \mid \Delta \vdash t : A \multimap B \qquad \Gamma \mid \Delta' \vdash u : A}{\Gamma \mid \Delta \cdot \Delta' \vdash t\,u : B} \qquad \frac{\Gamma \mid \Delta \cdot (x : A) \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\circ} x.\, t : A \multimap B}$$

$\approx$ Polakow & Pfenning's Non-Commutative Intuitionistic Linear Logic

$$\frac{\Gamma \mid \Delta \vdash t : A}{\Gamma \mid \Delta' \vdash t : A}$$ when Δ is a **subsequence** of Δ'

Remarks on the proof

To prove λ-definable $\subseteq$ star-free, we use:

Lemma (in our non-commutative λ-calculus)

For any purely affine A, *the closed terms of type* $A \multimap A$, *modulo* $=_\beta$, *form a* finite and aperiodic monoid.

(No semantics needed: syntax already finite!)

Conclude similarly to Hillebrand & Kanellakis.

Remarks on the proof

To prove λ-definable $\subseteq$ star-free, we use:

Lemma (in our non-commutative λ-calculus)

For any purely affine A, *the closed terms of type* $A \multimap A$, *modulo* $=_\beta$, *form a* finite and aperiodic monoid.

(No semantics needed: syntax already finite!)

Conclude similarly to Hillebrand & Kanellakis.

The converse is harder (unusual for implicit complexity!): how do we exploit the aperiodicity assumption?

Remarks on the proof

To prove λ-definable $\subseteq$ star-free, we use:

Lemma (in our non-commutative λ-calculus)

For any purely affine A, *the closed terms of type* $A \multimap A$, *modulo* $=_\beta$, *form a* finite and aperiodic monoid.

(No semantics needed: syntax already finite!)

Conclude similarly to Hillebrand & Kanellakis.

The converse is harder (unusual for implicit complexity!): how do we exploit the aperiodicity assumption?

Theorem (Krohn & Rhodes 1965 (special case))

Any aperiodic monoid can be "decomposed" as a wreath product *of "building blocks" which are certain monoids with 3 elements.*

Remarks on the proof

To prove λ-definable $\subseteq$ star-free, we use:

Lemma (in our non-commutative λ-calculus)

For any purely affine A, *the closed terms of type* $A \multimap A$, *modulo* $=_\beta$, *form a* finite and aperiodic monoid.

(No semantics needed: syntax already finite!)

Conclude similarly to Hillebrand & Kanellakis.

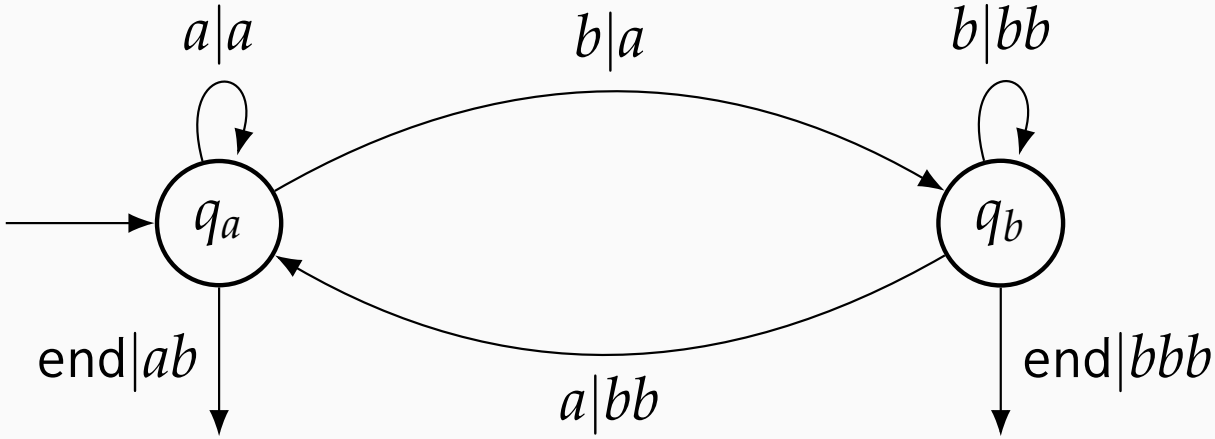
The converse is harder (unusual for implicit complexity!): how do we exploit the aperiodicity assumption?

Theorem (Krohn & Rhodes 1965 (special case))

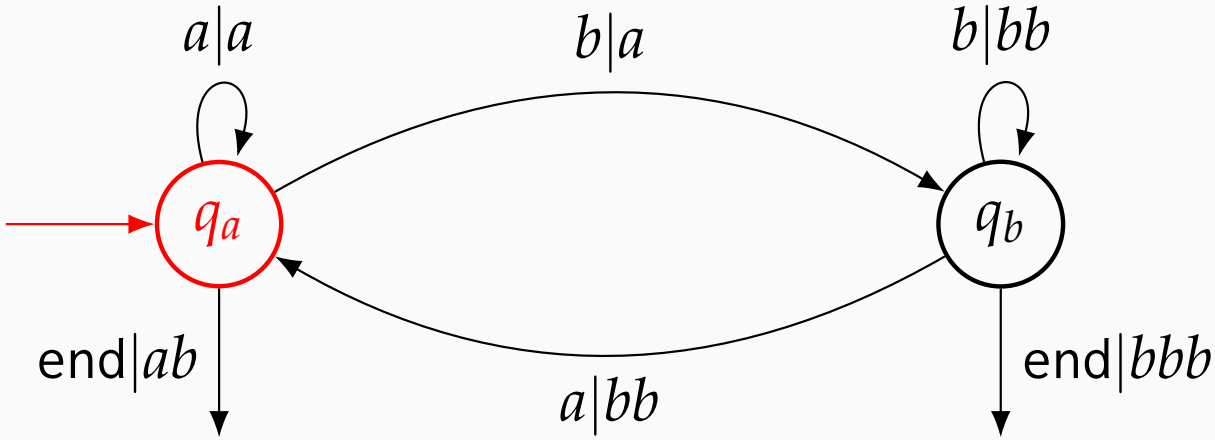
Any aperiodic monoid can be "decomposed" as a wreath product *of "building blocks" which are certain monoids with 3 elements.*

To avoid the scary algebra: a detour through *transducers*.

Sequential transducers: an example

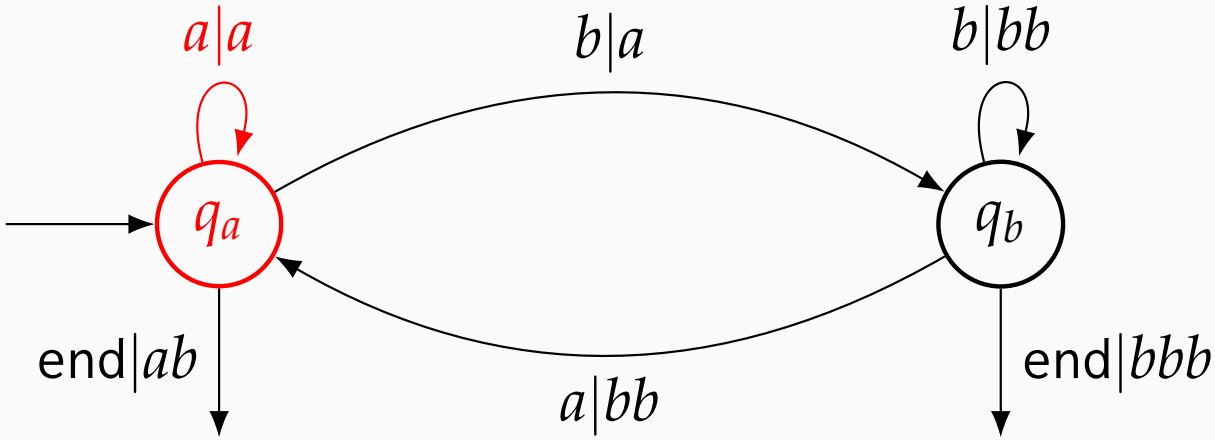


Sequential transducers: an example



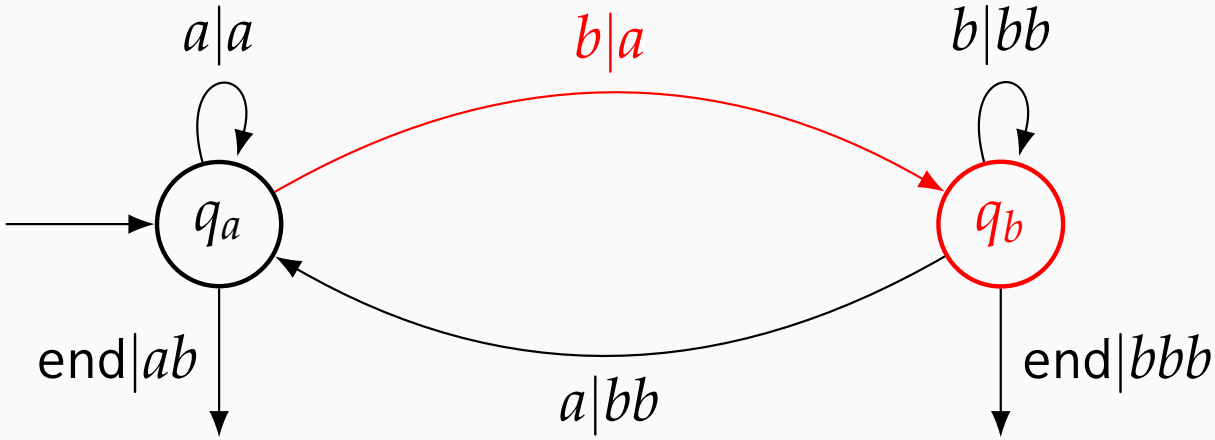
$abba \mapsto$

Sequential transducers: an example



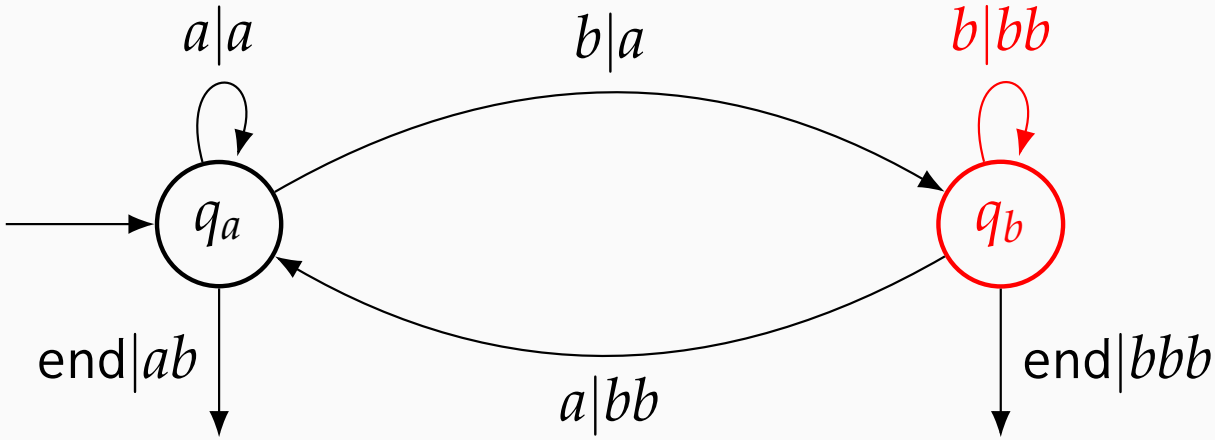
$abba \mapsto a$

Sequential transducers: an example



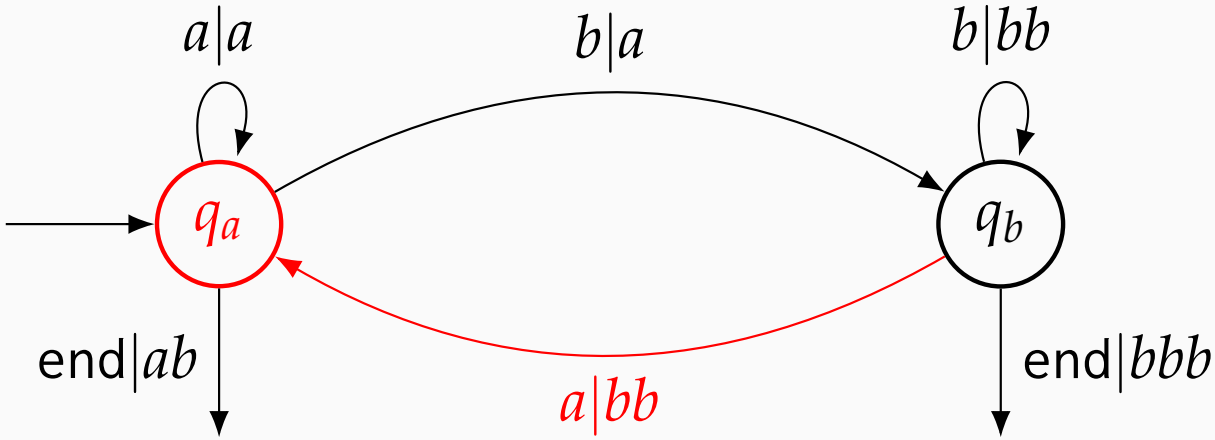
$abba \mapsto aa$

Sequential transducers: an example



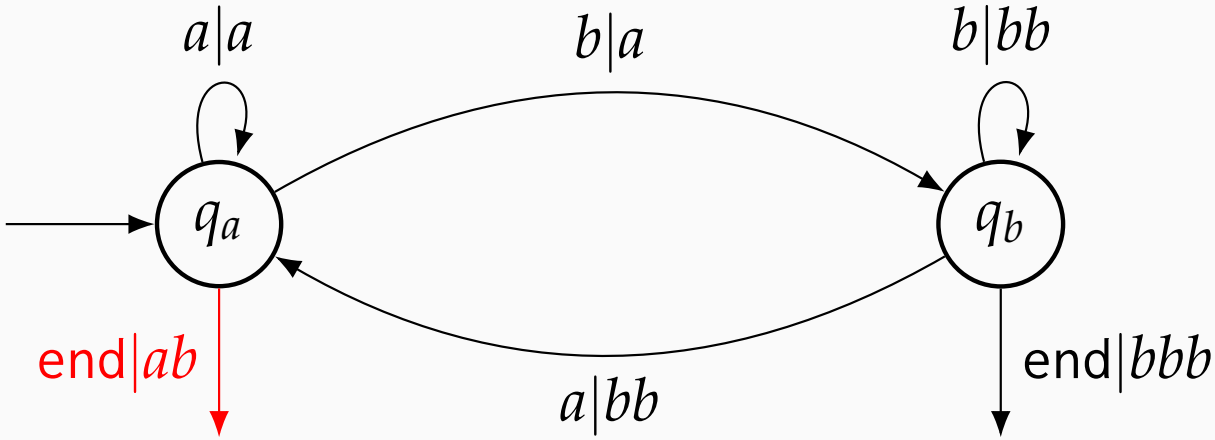
$abba \mapsto aabb$

Sequential transducers: an example



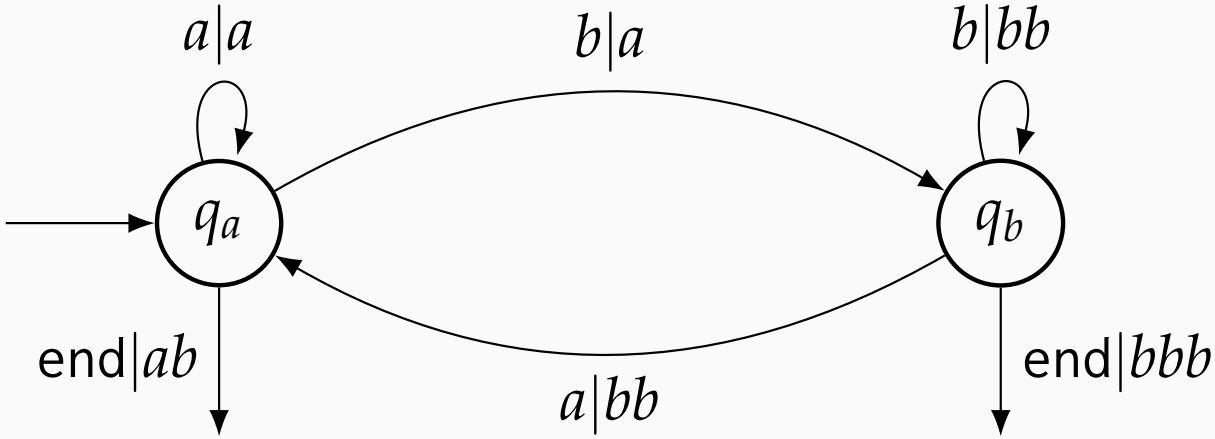
$$abba \mapsto aabbbb$$

Sequential transducers: an example



$$abba \mapsto aabbbbab$$

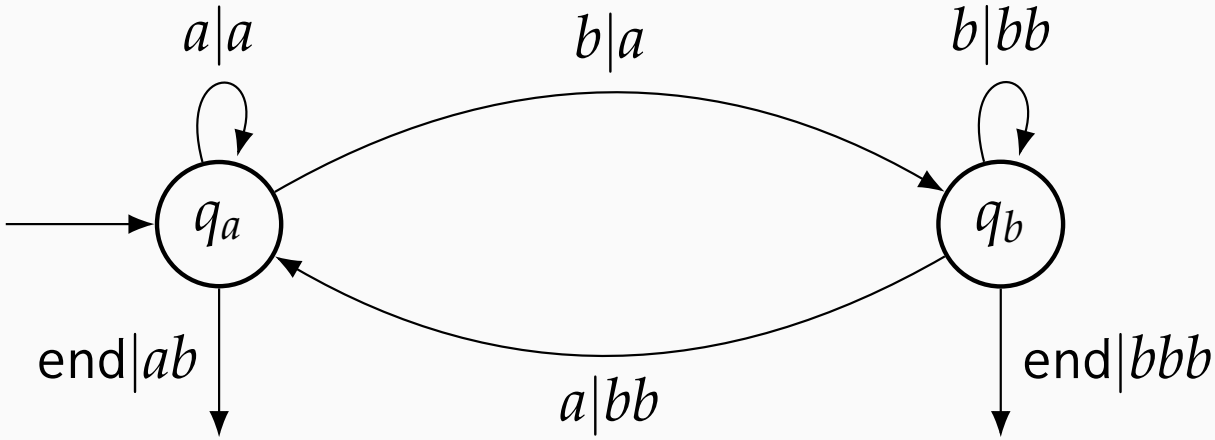
Sequential transducers: an example



$$abba \mapsto aabbbbab$$

$\longrightarrow$ double every b, prepend a, append b

Sequential transducers: an example

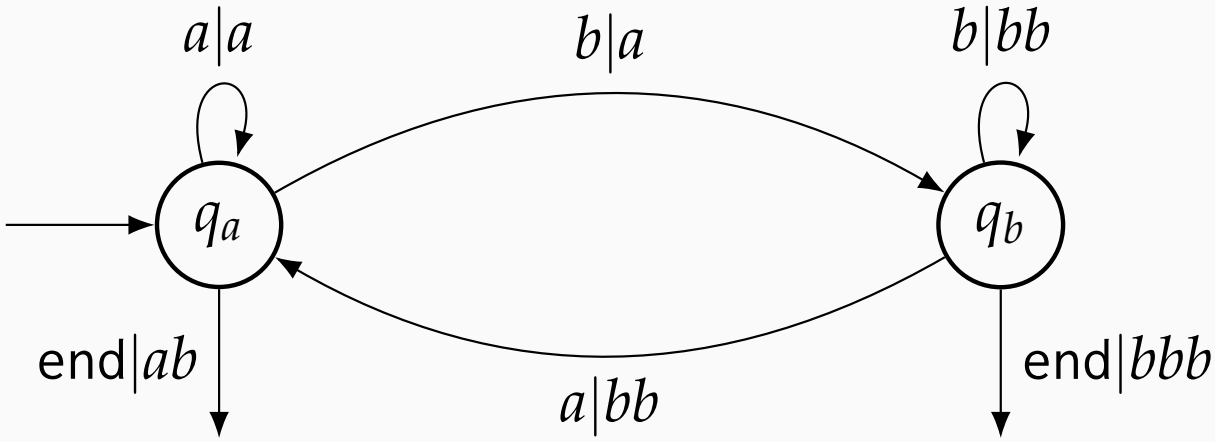


$$abba \mapsto aabbbbab$$

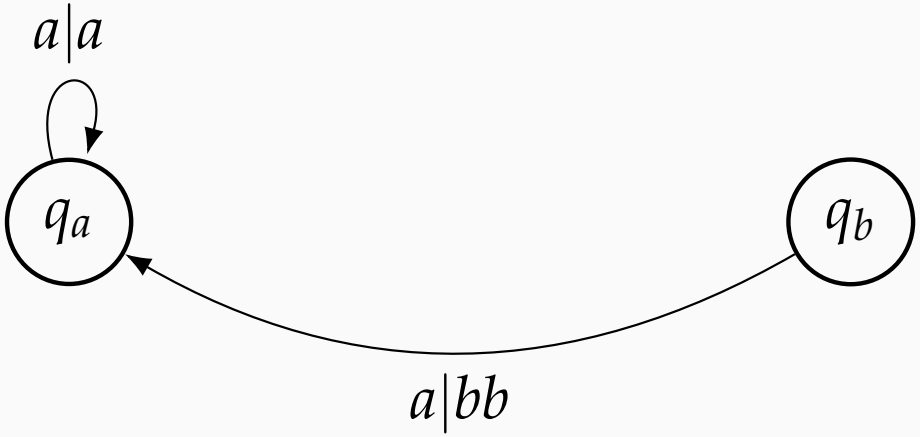
$\longrightarrow$ double every b, prepend a, append b

[DeYoung & Pfenning, APLAS'16: this $\iff$ cyclic proofs]

Aperiodic sequential transducers

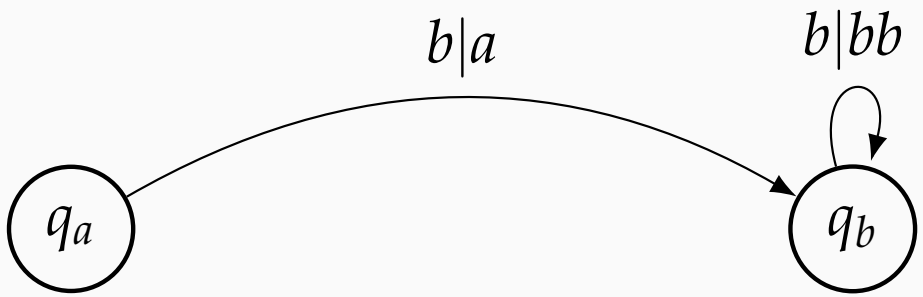


Aperiodic sequential transducers



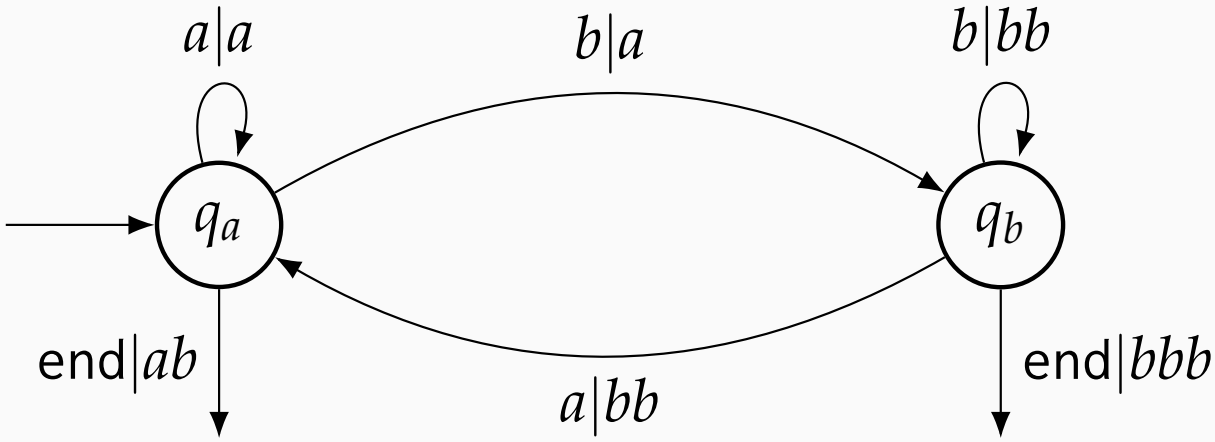
Transition for a: $\_ \mapsto q_a$

Aperiodic sequential transducers



Transition for a: $\_ \mapsto q_a$ Transition for b: $\_ \mapsto q_b$

Aperiodic sequential transducers



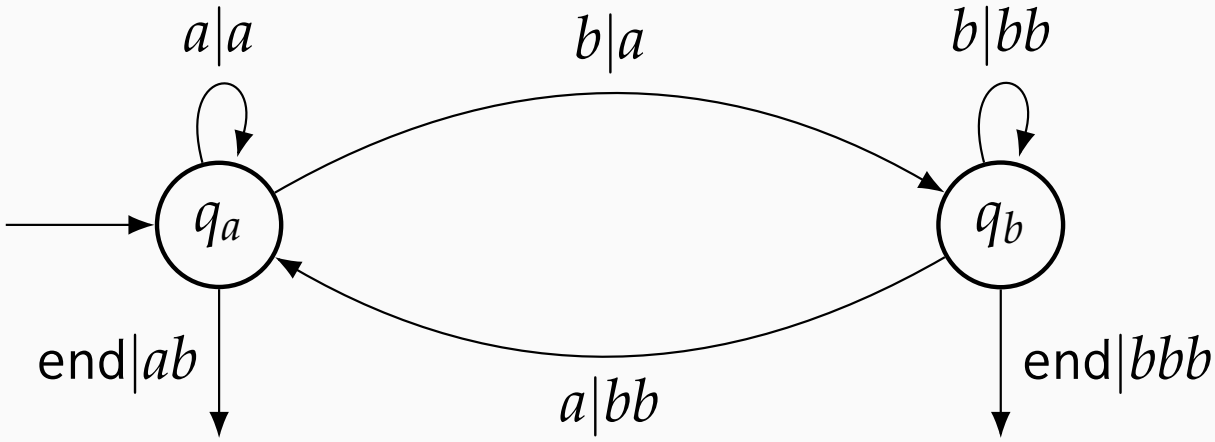
Transition for a: $\_ \mapsto q_a$ Transition for b: $\_ \mapsto q_b$

They generate the *transition monoid* $\{\mathrm{id}, (\_ \mapsto q_a), (\_ \mapsto q_b)\}$

(Remark: this 3-element monoid is the building block in Krohn–Rhodes!)

This monoid is aperiodic $\rightarrow$ *aperiodic sequential transducer*

Aperiodic sequential transducers



Transition for a: $\_ \mapsto q_a$ Transition for b: $\_ \mapsto q_b$

They generate the *transition monoid* $\{\mathrm{id}, (\_ \mapsto q_a), (\_ \mapsto q_b)\}$

(Remark: this 3-element monoid is the building block in Krohn–Rhodes!)

This monoid is aperiodic $\rightarrow$ *aperiodic sequential transducer*

Lemma

L star-free $\iff$ $L = f^{-1}(\varepsilon)$ for some aperiodic sequential f

The Krohn–Rhodes decomposition, again

Reformulation of the Krohn–Rhodes decomposition

Aperiodic sequential functions are generated by aper. seq. transducers *with 2 states* + usual function composition.

So it's enough to find λ-terms for transducers with 2 states.
(Not-so-trivial programming exercise!)

The Krohn–Rhodes decomposition, again

Reformulation of the Krohn–Rhodes decomposition

Aperiodic sequential functions are generated by aper. seq. transducers *with 2 states* + usual function composition.

So it's enough to find λ-terms for transducers with 2 states. (Not-so-trivial programming exercise!)

Theorem

Our non-commutative λ-calculus can define all aperiodic sequential functions with terms $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (A purely affine).

Corollary

It can define all star-free languages with $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Bool}[o]$.

String-to-string functions

A byproduct of our proof:

Theorem

Our non-commutative λ-calculus can define all aperiodic sequential functions with terms $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (A purely affine).

What about the converse?

String-to-string functions

A byproduct of our proof:

Theorem

Our non-commutative λ-calculus can define all aperiodic sequential functions with terms $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (A purely affine).

What about the converse?

False: we can code non-sequential functions, e.g. `reverse`.

- Exact characterization of $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$?

String-to-string functions

A byproduct of our proof:

Theorem

Our non-commutative λ-calculus can define all aperiodic sequential functions with terms $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (A purely affine).

What about the converse?

False: we can code non-sequential functions, e.g. `reverse`.

- Exact characterization of $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$?
- What happens in a commutative affine λ-calculus?
- $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma[o]$ in simply typed λ-calculus?

String-to-string functions

A byproduct of our proof:

Theorem

Our non-commutative λ-calculus can define all aperiodic sequential functions with terms $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (A purely affine).

What about the converse?

False: we can code non-sequential functions, e.g. `reverse`.

- Exact characterization of $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$?
- What happens in a commutative affine λ-calculus?
- $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma[o]$ in simply typed λ-calculus?
  We don't know, even for $\mathsf{Nat}[A] \to \mathsf{Nat}$ ($\Gamma = \Sigma = \{1\}$)
  Negative results by Statman in the 1980s (see also Joly 2001)

Regular functions

Theorem

$\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (*A purely affine*)
$\iff$ regular (*resp.* aperiodic regular) *functions*
in commutative (*resp. non-commutative*) *affine* λ*-calculus.*

$$\text{sequential fn} \subsetneq \text{rational fn} \subsetneq \textbf{regular fn}$$

Many characterizations; among others:

- *two-way* finite state transducers
- monadic second-order logic (aperiodic case: first-order)
- basic functions + combinators (several variants)
- copyless streaming string transducers

Regular functions

Theorem

$\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (*A purely affine*)
$\iff$ regular (*resp.* aperiodic regular) *functions*
in commutative (*resp. non-commutative*) *affine* λ*-calculus.*

$$\text{sequential fn} \subsetneq \text{rational fn} \subsetneq \textbf{regular fn}$$

Many characterizations; among others:

- *two-way* finite state transducers
- monadic second-order logic (aperiodic case: first-order)
- basic functions + combinators (several variants)
- **copyless** streaming string transducers
  - Sounds suspiciously like affine types!

Regular functions

Theorem

$\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ (*A purely affine*)
$\iff$ regular (*resp.* aperiodic regular) *functions*
in commutative (*resp. non-commutative*) *affine* λ*-calculus.*

$$\text{sequential fn} \subsetneq \text{rational fn} \subsetneq \textbf{regular fn}$$

Many characterizations; among others:

- *two-way* finite state transducers (2DFT)
  - also, affine λ-calculus $\subseteq$ 2DFT via *geometry of interaction*
- monadic second-order logic (aperiodic case: first-order)
- basic functions + combinators (several variants)
- **copyless** streaming string transducers
  - Sounds suspiciously like affine types!

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\mathsf{out} = XY$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\mathsf{out} = XY$

Execution over *abaa*: start with

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

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*:

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

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\mathtt{out} = XY$

Execution over $abaa$:

$$X = ab \qquad Y = ba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\mathtt{out} = XY$

Execution over $aba\color{red}{a}a$:

$$X = aba \qquad Y = aba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*:

$$X = abaa \qquad Y = aaba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\mathtt{out} = XY$

Execution over *abaa*: $f(abaa) = abaaaaba$

$$X = abaa \qquad Y = aaba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\texttt{out} = XY$

Execution over *abaa*: $f(abaa) = abaaaaba, f : w \mapsto w \cdot \texttt{reverse}(w)$

$$X = abaa \qquad Y = aaba$$

Streaming string transducers

- Finite set of Σ^*-valued *registers* e.g. $R = \{X, Y\}$
- Initial values $R \to \Sigma^*$ e.g. $X_{\text{init}} = Y_{\text{init}} = \varepsilon$
- *Register update function* e.g.

$$a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \qquad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$$

- "output function" e.g. $\texttt{out} = XY$

Execution over *abaa*: $f(abaa) = abaaaaba, f : w \mapsto w \cdot \texttt{reverse}(w)$

$$X = abaa \qquad Y = aaba$$

SSTs can also have *states*: their memory is $Q \times (\Sigma^*)^R$ ($|Q| < \infty$)

Equivalent to so-called "HDT0L systems"

Copyless streaming string transducers

Copyless SSTs explained with linear logic connectives:

- Consider the memory as $M = Q \otimes \Sigma^* \otimes \ldots \otimes \Sigma^*$
  $(Q \cong 1 \oplus \ldots \oplus 1,$
  $\mathsf{concat} : \Sigma^* \otimes \Sigma^* \multimap \Sigma^*)$
- *Affine* transitions $M \multimap M$, e.g. $(X := Xa,\ Y := aY)$
  each register appears only once on the right of ':='

Theorem (Alur & Černý 2010)

The functions computed by copyless SSTs are the regular functions.

No '$\oplus$' in our affine λ-calculus $\longrightarrow$ rely on Krohn–Rhodes

Copyless streaming string transducers

Copyless SSTs explained with linear logic connectives:

- Consider the memory as $M = Q \otimes \Sigma^* \otimes \ldots \otimes \Sigma^*$
  $(Q \cong 1 \oplus \ldots \oplus 1,$
  $\mathsf{concat} : \Sigma^* \otimes \Sigma^* \multimap \Sigma^*)$
- *Affine* transitions $M \multimap M$, e.g. $(X := Xa,\ Y := aY)$
  each register appears only once on the right of ':='

Theorem (Alur & Černý 2010)

The functions computed by copyless SSTs are the regular functions.

No '$\oplus$' in our affine λ-calculus $\longrightarrow$ rely on Krohn–Rhodes

Next: linear logic helps us understand automata

Additive connectives for streaming tree transducers

Copyless streaming *tree* transducers $\subset$ regular *tree* functions;
conjectured to be a *strict inclusion*.

To recover an equality:
ad-hoc relaxation called "single use restriction".

Principled explanation via linear logic:
keep $M \multimap M$, just add the *additive conjunction*!

e.g. $\qquad M = Q \otimes \Sigma^* \otimes (\Sigma^* \,\&\, \Sigma^*)$

Additive connectives for streaming tree transducers

Copyless streaming *tree* transducers $\subset$ regular *tree* functions;
conjectured to be a *strict inclusion*.

To recover an equality:
ad-hoc relaxation called "single use restriction".

Principled explanation via linear logic:
keep $M \multimap M$, just add the *additive conjunction*!

e.g. $\quad M = Q \otimes \Sigma^* \otimes (\Sigma^* \,\&\, \Sigma^*)$

Also, *category-theoretic* justification: memories / transitions
should be objects / morphisms in a nice category (next slide).

(see also: Colcombet & Petrişan's categorical automata theory)

Additive connectives for streaming tree transducers

Copyless streaming *tree* transducers $\subset$ regular *tree* functions;
conjectured to be a *strict inclusion*.

To recover an equality:
ad-hoc relaxation called "single use restriction".

Principled explanation via linear logic:
keep $M \multimap M$, just add the *additive conjunction*!

$$\text{e.g.} \qquad M = Q \otimes \Sigma^* \otimes (\Sigma^* \,\&\, \Sigma^*) = \bigoplus_{q \in Q} \Sigma^* \otimes (\Sigma^* \,\&\, \Sigma^*)$$

Also, *category-theoretic* justification: memories / transitions
should be objects / morphisms in a nice category (next slide).

(see also: Colcombet & Petrişan's categorical automata theory)

A category of single-use-restricted register assignments

We can build a category whose objects are morally

$$\bigoplus_{u \in U} \&_{x \in X(u)} (\Sigma^* \otimes \ldots [n(u,x) \text{ times}] \ldots \otimes \Sigma^*)$$

with $A \otimes (B \,\&\, C) \equiv (A \otimes B) \,\&\, (A \otimes C)$, $A \,\&\, (B \oplus C) \equiv (A \,\&\, B) \oplus (A \,\&\, C)$

$\cong$ dependent variant of de Paiva's *Dialectica* categories **DC**,
think $\exists u \forall x$ (also related to *lenses* / bidirectional programming)

A category of single-use-restricted register assignments

We can build a category whose objects are morally

$$\bigoplus_{u \in U} \&_{x \in X(u)} (\Sigma^* \otimes \ldots [n(u,x) \text{ times}] \ldots \otimes \Sigma^*)$$

with $A \otimes (B \,\&\, C) \equiv (A \otimes B) \,\&\, (A \otimes C)$, $A \,\&\, (B \oplus C) \equiv (A \,\&\, B) \oplus (A \,\&\, C)$

$\cong$ dependent variant of de Paiva's *Dialectica* categories **DC**, think $\exists u \forall x$ (also related to *lenses* / bidirectional programming)

Theorem (Dialectica + combinatorics on strings)

This category is monoidal closed (but wouldn't be without '&').

- used to prove affine λ-calculus $\subseteq$ regular functions: *semantic evaluation*, as for STLC $\subseteq$ regular languages

A category of single-use-restricted register assignments

We can build a category whose objects are morally

$$\bigoplus_{u \in U} \&_{x \in X(u)} (\Sigma^* \otimes \ldots [n(u,x) \text{ times}] \ldots \otimes \Sigma^*)$$

with $A \otimes (B \,\&\, C) \equiv (A \otimes B) \,\&\, (A \otimes C)$, $A \,\&\, (B \oplus C) \equiv (A \,\&\, B) \oplus (A \,\&\, C)$

$\cong$ dependent variant of de Paiva's *Dialectica* categories **DC**,
think $\exists u \forall x$ (also related to *lenses* / bidirectional programming)

Theorem (Dialectica + combinatorics on strings)

This category is monoidal closed (but wouldn't be without '&').

- used to prove affine λ-calculus $\subseteq$ regular functions:
  semantic evaluation, as for STLC $\subseteq$ regular languages
- explains why regular functions are *closed under composition*
  (subtle point, wrong proof in original copyless SST paper)

A category of single-use-restricted register assignments

We can build a category whose objects are morally

$$\bigoplus_{u \in U} \&_{x \in X(u)} (\Sigma^* \otimes \ldots [n(u,x) \text{ times}] \ldots \otimes \Sigma^*)$$

with $A \otimes (B \,\&\, C) \equiv (A \otimes B) \,\&\, (A \otimes C)$, $A \,\&\, (B \oplus C) \equiv (A \,\&\, B) \oplus (A \,\&\, C)$

$\cong$ dependent variant of de Paiva's *Dialectica* categories **DC**,
think $\exists u \forall x$ (also related to *lenses* / bidirectional programming)

Theorem (Dialectica + combinatorics on strings)

This category is monoidal closed (but wouldn't be without '&').

- used to prove affine λ-calculus $\subseteq$ regular functions:
  semantic evaluation, as for STLC $\subseteq$ regular languages
- explains why regular functions are *closed under composition*
  (subtle point, wrong proof in original copyless SST paper)
- elimination of '&' in SST $\cong$ determinization

Conclusion (1): summary

$\mathsf{Str}_\Sigma[A] \multimap \mathsf{Bool}[o]$ / $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ correspond to:

non-commutative λ-calculus	star-free languages, aperiodic reg. fun.
(commutative) affine λ-calculus	regular functions

- Succesful revival of ideas from the 1990s: automata vs λ-calculus, non-commutative type systems
- Connections between linear logic / categorical semantics and modern theory of transducers

Conclusion (1): summary

$\mathsf{Str}_\Sigma[A] \multimap \mathsf{Bool}[o]$ / $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ correspond to:

non-commutative λ-calculus	star-free languages, aperiodic reg. fun.
(commutative) affine λ-calculus	regular functions

- Succesful revival of ideas from the 1990s: automata vs λ-calculus, non-commutative type systems
- Connections between linear logic / categorical semantics and modern theory of transducers
- Not covered here:
  - $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma[o]$ with affine types ($\rightsquigarrow$ subclass of *polyregular functions* [Bojańczyk 2018])
  - Aperiodic regular fn vs planar geometry of interaction $\longrightarrow$ variant of SSTs using planar diagrams

Conclusion (2): future work

non-commutative λ-calculus	star-free languages, aperiodic reg. fun.
(commutative) affine λ-calculus	regular functions

- What about $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma[o]$ in simply typed λ-calculus?
  - Hillebrand–Kanellakis $\implies$ f^{-1}(reg. lang.) is regular
  - Contain at least copyful SSTs + composition
  - Possible connection with collapsible pushdown automata

Conclusion (2): future work

non-commutative λ-calculus	star-free languages, aperiodic reg. fun.
(commutative) affine λ-calculus	regular functions

- What about $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma[o]$ in simply typed λ-calculus?
  - Hillebrand–Kanellakis $\implies$ f^{-1}(reg. lang.) is regular
  - Contain at least copyful SSTs + composition
  - Possible connection with collapsible pushdown automata
- Revisit systems from implicit complexity
  - regular functions in 2nd-order Elementary Affine Logic via proof nets (j.w.w. Pistone, Seiller, Tortora de Falco)
  - parsimonious λ-calculus (Mazza) vs polyregular functions?

Conclusion (2): future work

non-commutative λ-calculus	star-free languages, aperiodic reg. fun.
(commutative) affine λ-calculus	regular functions

- What about $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma[o]$ in simply typed λ-calculus?
  - Hillebrand–Kanellakis $\implies$ f^{-1}(reg. lang.) is regular
  - Contain at least copyful SSTs + composition
  - Possible connection with collapsible pushdown automata
- Revisit systems from implicit complexity
  - regular functions in 2nd-order Elementary Affine Logic via proof nets (j.w.w. Pistone, Seiller, Tortora de Falco)
  - parsimonious λ-calculus (Mazza) vs polyregular functions?
- Linearity in infinite alphabets / nominal sets?
  - Single use automata [Bojańczyk & Stefański, ICALP 2020]