Implicit automata in typed λ-calculi

Pierre Pradic — pierre.pradic@cs.ox.ac.uk
University of Oxford

j.w.w. Nguyễn Lê Thành Dũng (a.k.a. Tito) — nltd@nguyentito.eu
Laboratoire d'informatique de Paris Nord

MIMUW Automata seminar, June 24th, 2020

The idea

Many equivalent definitions of regular languages:
regexps, automata (deterministic or not), MSO, ...

The idea

Many equivalent definitions of regular languages:
regexps, automata (deterministic or not), MSO, **programming languages**, ...

Implicit complexity: machine-free characterizations of complexity classes
using high-level programming languages

Big project: the same thing for automata instead of complexity

programming paradigm	declarative	functional
complexity classes	descriptive complexity	implicit complexity
automata theory	subsystems of MSO	**this work**

The idea

Many equivalent definitions of regular languages:
regexps, automata (deterministic or not), MSO, **programming languages**, ...

Implicit complexity: machine-free characterizations of complexity classes using high-level programming languages

Big project: the same thing for automata instead of complexity

programming paradigm	declarative	functional
complexity classes	descriptive complexity	implicit complexity
automata theory	subsystems of MSO	**this work**

Our starting point: Hillebrand & Kanellakis's theorem (1996)

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

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for e.g. Haskell

The idea

Many equivalent definitions of regular languages:
regexps, automata (deterministic or not), MSO, **programming languages**, ...

Implicit complexity: machine-free characterizations of complexity classes using high-level programming languages

Big project: the same thing for automata instead of complexity

programming paradigm	declarative	functional
complexity classes	descriptive complexity	implicit complexity
automata theory	subsystems of MSO	**this work**

Our starting point: Hillebrand & Kanellakis's theorem (1996)

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

Typed λ-calculi ($\cong$ constructive logics): theoretical basis for e.g. Haskell

Other "implicit automata" results: cf. MIMUW automata seminar 2020-05-13
DeYoung & Pfenning 2016 / Kuperberg et al. 2019, using *circular proofs*

The λ-calculus

λ-terms: meant to be a naive syntactic theory of functions

$$t, u ::= x \mid t\ u \mid \lambda x.\ t \qquad\qquad f\ t \approx f(t) \qquad \lambda x.\ t \approx (x \mapsto t)$$

Notational conventions: $t\ u\ v = (t\ u)\ v$ and $\lambda x.\ t\ u = \lambda x.\ (t\ u)$

The λ-calculus

λ-terms: meant to be a naive syntactic theory of functions

$$t, u ::= x \mid t\ u \mid \lambda x.\ t \qquad\qquad f\ t \ \approx\ f(t) \qquad \lambda x.\ t \ \approx\ (x \mapsto t)$$

Notational conventions: $t\ u\ v = (t\ u)\ v$ and $\lambda x.\ t\ u = \lambda x.\ (t\ u)$

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

Operational semantics: program execution by rewriting

$$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}\ (t \text{ with } x \textit{ substituted} \text{ by } u)$$

We write $=_\beta$ for the smallest equivalence relation containing $\longrightarrow_\beta$

The λ-calculus

λ-terms: meant to be a naive syntactic theory of functions

$$t, u ::= x \mid t\ u \mid \lambda x.\ t \qquad f\ t \approx f(t) \qquad \lambda x.\ t \approx (x \mapsto t)$$

Notational conventions: $t\ u\ v = (t\ u)\ v$ and $\lambda x.\ t\ u = \lambda x.\ (t\ u)$

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

Operational semantics: program execution by rewriting

$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}$ (t with x *substituted* by u)

We write $=_\beta$ for the smallest equivalence relation containing $\longrightarrow_\beta$

No primitive data; encodings using functions

Useful example: booleans

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

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

The λ-calculus

λ-terms: meant to be a naive syntactic theory of functions

$$t, u ::= x \mid t\ u \mid \lambda x.\ t \qquad f\ t \approx f(t) \qquad \lambda x.\ t \approx (x \mapsto t)$$

Notational conventions: $t\ u\ v = (t\ u)\ v$ and $\lambda x.\ t\ u = \lambda x.\ (t\ u)$

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

Operational semantics: program execution by rewriting

$(\lambda x.\ t)\ u \longrightarrow_\beta t\{x := u\}$ (t with x *substituted* by u)

We write $=_\beta$ for the smallest equivalence relation containing $\longrightarrow_\beta$

No primitive data; encodings using functions

Useful example: booleans

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

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

Also encodings of natural numbers, strings... (later!)

Untyped λ-calculus is Turing-complete

The simply typed λ-calculus

Simple types: specifications for λ-calculus programs

$$A, B ::= o \text{ (base type)} \mid A \to B \text{ (functions from } A \text{ to } B)$$

Convention: $A \to B \to C = A \to (B \to C)$

$t : A$ ("t is of type A") defined by induction on the syntax

The simply typed λ-calculus

Simple types: specifications for λ-calculus programs

$$A, B ::= o \text{ (base type)} \mid A \to B \text{ (functions from } A \text{ to } B\text{)}$$

Convention: $A \to B \to C = A \to (B \to C)$

$t : A$ ("t is of type A") defined by induction on the syntax

Typed booleans

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

so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$ (for t *closed*, i.e. w/ no free vars)

The simply typed λ-calculus

Simple types: specifications for λ-calculus programs

$$A, B ::= o \text{ (base type)} \mid A \to B \text{ (functions from } A \text{ to } B\text{)}$$

Convention: $A \to B \to C = A \to (B \to C)$

$t : A$ ("t is of type A") defined by induction on the syntax

Typed booleans

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

so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$ (for t *closed*, i.e. w/ no free vars)

Pro: all well-typed programs *terminate* Con: loss of Turing-completeness

The simply typed λ-calculus

Simple types: specifications for λ-calculus programs

$$A, B ::= o \text{ (base type)} \mid A \to B \text{ (functions from } A \text{ to } B)$$

Convention: $A \to B \to C = A \to (B \to C)$
$t : A$ ("t is of type A") defined by induction on the syntax

Typed booleans

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

so that $t : \texttt{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$ (for t *closed*, i.e. w/ no free vars)

Pro: all well-typed programs *terminate* Con: loss of Turing-completeness

Church encoding of strings

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

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

The simply typed λ-calculus

Simple types: specifications for λ-calculus programs

$$A, B ::= o \text{ (base type)} \mid A \to B \text{ (functions from } A \text{ to } B\text{)}$$

Convention: $A \to B \to C = A \to (B \to C)$
$t : A$ ("t is of type A") defined by induction on the syntax

Typed booleans

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

so that $t : \mathsf{Bool} \iff t =_\beta \texttt{true}$ or $t =_\beta \texttt{false}$ (for t *closed*, i.e. w/ no free vars)

Pro: all well-typed programs *terminate* Con: loss of Turing-completeness

Church encoding of strings

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

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

Languages: $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$ (A is any simple type)
(for $w \in \Sigma^*$, $\overline{w} : \mathsf{Str}_\Sigma$ therefore $\overline{w} : \mathsf{Str}_\Sigma\{o := A\}$)

Regular languages in the simply typed λ-calculus

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *is* regular $\iff$ *L is defined by some* $t : \mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$ *in the simply typed λ-calculus*

(A: *arbitrary simple type, may depend on L*)

Regular languages in the simply typed λ-calculus

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *is* regular $\iff$ *L is defined by some* $t : \mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$ *in the simply typed λ-calculus*

(*A: arbitrary simple type, may depend on L*)

Proof ideas for non-trivial direction ($\Longleftarrow$):

- $w \in \Sigma^* \mapsto \overline{w} : \{t \mid t : \mathsf{Str}_\Sigma\{o := A\}\}/=_\beta$ is a *monoid morphism*

  (recall that $\overline{w} \approx (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$)

Regular languages in the simply typed λ-calculus

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *is* regular $\iff$ *L is defined by some* $t : \mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$ *in the simply typed λ-calculus*

(*A: arbitrary simple type, may depend on L*)

Proof ideas for non-trivial direction ($\Longleftarrow$):

- $w \in \Sigma^* \mapsto \overline{w} : \{t \mid t : \mathsf{Str}_\Sigma\{o := A\}\}/=_\beta$ is a *monoid morphism*
  (recall that $\overline{w} \approx (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$)
- To get a *finite monoid* as target, evaluate in a *finite semantics*
  $w \in \Sigma^* \mapsto [\![\overline{w}]\!] \in [\![\mathsf{Str}_\Sigma\{o := A\}]\!]$, with $|\,[\![\mathsf{Str}_\Sigma\{o := A\}]\!]\,| < \infty$

Regular languages in the simply typed λ-calculus

Theorem (Hillebrand & Kanellakis 1996)

$L \subseteq \Sigma^*$ *is* regular $\iff$ *L is defined by some* $t : \mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$ *in the simply typed λ-calculus*

(*A: arbitrary simple type, may depend on L*)

Proof ideas for non-trivial direction ($\Longleftarrow$):

- $w \in \Sigma^* \mapsto \overline{w} : \{t \mid t : \mathsf{Str}_\Sigma\{o := A\}\}/=_\beta$ is a *monoid morphism*
  (recall that $\overline{w} \approx (f_a, f_b) \mapsto f_{w[1]} \circ \ldots \circ f_{w[n]}$)
- To get a *finite monoid* as target, evaluate in a *finite semantics*
  $w \in \Sigma^* \mapsto [\![\overline{w}]\!] \in [\![\mathsf{Str}_\Sigma\{o := A\}]\!]$, with $|\,[\![\mathsf{Str}_\Sigma\{o := A\}]\!]\,| < \infty$

Canonical semantics: realizes the naive interpretation

- for *types*: $[\![o]\!] =$ some 2-element set, $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$
- for *terms*: $t : A \rightsquigarrow [\![t]\!] \in [\![A]\!]$ inductively defined, e.g. $[\![f\ x]\!] = [\![f]\!]\,([\![x]\!])$
- soundness: $t =_\beta u \implies [\![t]\!] = [\![u]\!]$
- preserves some relevant information: $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$

String-to-string functions

Open problem (that we don't solve :-( )

Nice characterization (preferably automata-theoretic) of
$\mathsf{Str}_\Gamma\{o := A\} \to \mathsf{Str}_\Sigma$ in the simply typed λ-calculus?

($\mathsf{Str}_\Gamma \to \mathsf{Str}_\Sigma$ described by base functions + composition [Zaionc 1987])

- Regular languages closed under inverse image of such functions
- Closed under compostion
- May have hyperexponential growth

String-to-string functions

Open problem (that we don't solve :-( )

Nice characterization (preferably automata-theoretic) of
$\mathsf{Str}_\Gamma\{o := A\} \to \mathsf{Str}_\Sigma$ in the simply typed λ-calculus?

($\mathsf{Str}_\Gamma \to \mathsf{Str}_\Sigma$ described by base functions + composition [Zaionc 1987])

- Regular languages closed under inverse image of such functions
- Closed under compostion
- May have hyperexponential growth

Lower bound (we may come back to this at the end):

- contains HDT0L transductions ($\cong$ copyful SST)
- HDT0L+composition = higher-order pushdown transducers
  (unproven claim in [Sénizergues 2007])
- in particular, contains all polyregular functions

String-to-string functions

Open problem (that we don't solve :-( )

Nice characterization (preferably automata-theoretic) of
$\mathsf{Str}_\Gamma\{o := A\} \to \mathsf{Str}_\Sigma$ in the simply typed λ-calculus?

($\mathsf{Str}_\Gamma \to \mathsf{Str}_\Sigma$ described by base functions + composition [Zaionc 1987])

- Regular languages closed under inverse image of such functions
- Closed under compostion
- May have hyperexponential growth

Lower bound (we may come back to this at the end):

- contains HDT0L transductions ($\cong$ copyful SST)
- HDT0L+composition = higher-order pushdown transducers
  (unproven claim in [Sénizergues 2007])
- in particular, contains all polyregular functions

Next: more restrictive type systems $\rightsquigarrow$ smaller classes of transductions
+ one smaller class of languages

actual results instead of open problems

Reminder: star-free languages and aperiodic monoids

Our goal now: characterize *star-free languages* (ICALP 2020 paper).

Definition

A language is *star-free* if is defined by some regexp *without repetition star* L^*, but potentially with *complementation* $L^c = \Sigma^* \setminus L$.

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

Reminder: star-free languages and aperiodic monoids

Our goal now: characterize *star-free languages* (ICALP 2020 paper).

Definition

A language is *star-free* if is defined by some regexp *without repetition star* L^*, but potentially with *complementation* $L^c = \Sigma^* \setminus L$.

Example with $\Sigma = \{a, b\}$: $(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa\varnothing^c | \varnothing^c bb\varnothing^c)^c$
However $(aa)^*$ is *not* star-free...

Reminder: star-free languages and aperiodic monoids

Our goal now: characterize *star-free languages* (ICALP 2020 paper).

Definition

A language is *star-free* if is defined by some regexp *without repetition star* L^*, but potentially with *complementation* $L^c = \Sigma^* \setminus L$.

Example with $\Sigma = \{a, b\}$: $(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa \varnothing^c | \varnothing^c bb \varnothing^c)^c$
However $(aa)^*$ is *not* star-free...

Theorem (Schützenberger 1965)

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

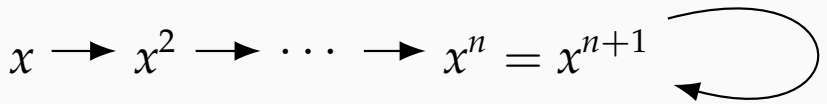
Reminder: star-free languages and aperiodic monoids

Our goal now: characterize *star-free languages* (ICALP 2020 paper).

Definition

A language is *star-free* if is defined by some regexp *without repetition star* L^*, but potentially with *complementation* $L^c = \Sigma^* \setminus L$.

Example with $\Sigma = \{a, b\}$: $(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa\varnothing^c | \varnothing^c bb\varnothing^c)^c$
However $(aa)^*$ is *not* star-free...



Definition

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

Theorem (Schützenberger 1965)

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

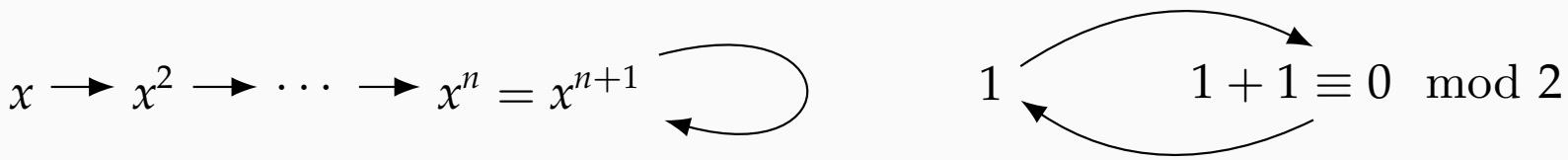
Reminder: star-free languages and aperiodic monoids

Our goal now: characterize *star-free languages* (ICALP 2020 paper).

Definition

A language is *star-free* if is defined by some regexp *without repetition star* L^*, but potentially with *complementation* $L^c = \Sigma^* \setminus L$.

Example with $\Sigma = \{a, b\}$: $(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa\varnothing^c | \varnothing^c bb\varnothing^c)^c$
However $(aa)^*$ is *not* star-free... its syntactic monoid is $\mathbb{Z}/(2)$



Definition

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

Theorem (Schützenberger 1965)

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

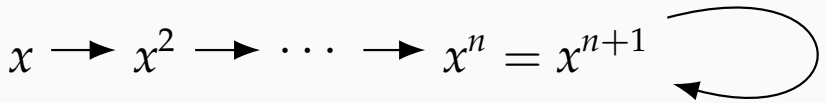
Reminder: star-free languages and aperiodic monoids

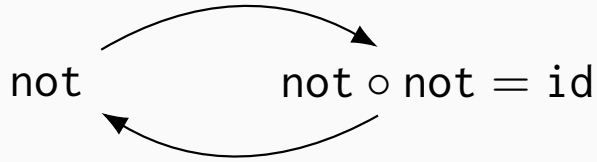
Our goal now: characterize *star-free languages* (ICALP 2020 paper).

Definition

A language is *star-free* if is defined by some regexp *without repetition star* L^*, but potentially with *complementation* $L^c = \Sigma^* \setminus L$.

Example with $\Sigma = \{a, b\}$: $(ab)^* = (b\varnothing^c | \varnothing^c a | \varnothing^c aa\varnothing^c | \varnothing^c bb\varnothing^c)^c$
However $(aa)^*$ is *not* star-free... its syntactic monoid is $\mathbb{Z}/(2) \cong \{\mathtt{id}_{\mathsf{Bool}}, \mathtt{not}\}$





Definition

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

Theorem (Schützenberger 1965)

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

From aperiodicity to non-commutative affine types

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

$\rightsquigarrow$ must exclude $\mathtt{not} : \mathsf{Bool} \to \mathsf{Bool}$ which merely permutes arguments!

$$\text{for } b : \mathsf{Bool},\ b\ x\ y \approx \mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \quad \rightsquigarrow \quad \mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

From aperiodicity to non-commutative affine types

How to enforce aperiodicity in a typed λ-calculus? $\mathtt{not}^{n+1} \neq \mathtt{not}^{n}$
$\rightsquigarrow$ must exclude $\mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}$ which merely permutes arguments!

$$\text{for } b : \mathtt{Bool},\ b\ x\ y \approx \mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \quad \rightsquigarrow \quad \mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

Idea 1: non-commutative typing

Make the order of arguments matter: "a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should not use x after y, since $\lambda x.$ occurs before λy."

From aperiodicity to non-commutative affine types

How to enforce aperiodicity in a typed λ-calculus? $\mathtt{not}^{n+1} \neq \mathtt{not}^{n}$
$\rightsquigarrow$ must exclude $\mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}$ which merely permutes arguments!

$$\text{for } b : \mathtt{Bool},\ b\ x\ y \approx \mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \quad \rightsquigarrow \quad \mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

Idea 1: non-commutative typing

Make the order of arguments matter: "a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should not use x after y, since $\lambda x.$ occurs before λy."

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

From aperiodicity to non-commutative affine types

How to enforce aperiodicity in a typed λ-calculus? $\mathtt{not}^{n+1} \neq \mathtt{not}^{n}$
$\rightsquigarrow$ must exclude $\mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}$ which merely permutes arguments!

$$\text{for } b : \mathtt{Bool},\ b\ x\ y \approx \mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \quad \rightsquigarrow \quad \mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

Idea 1: non-commutative typing

Make the order of arguments matter: "a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should not use x after y, since $\lambda x.$ occurs before λy."

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

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y) \quad \text{which has a } y \text{ occuring before an } x$$

From aperiodicity to non-commutative affine types

How to enforce aperiodicity in a typed λ-calculus? $\mathtt{not}^{n+1} \neq \mathtt{not}^{n}$
$\rightsquigarrow$ must exclude $\mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}$ which merely permutes arguments!

$$\text{for } b : \mathtt{Bool},\ b\ x\ y \approx \mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \quad \rightsquigarrow \quad \mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

Idea 1: non-commutative typing

Make the order of arguments matter: "a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should not use x after y, since $\lambda x.$ occurs before λy."

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

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y) \quad \text{which has a } y \text{ occuring before an } x$$

Caused by *2 occurrences* of z that led to *duplication* of $(x\ y)$

From aperiodicity to non-commutative affine types

How to enforce aperiodicity in a typed λ-calculus? $\mathtt{not}^{n+1} \neq \mathtt{not}^{n}$
$\rightsquigarrow$ must exclude $\mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}$ which merely permutes arguments!

$$\text{for } b : \mathtt{Bool},\ b\ x\ y \approx \mathtt{if}\ b\ \mathtt{then}\ x\ \mathtt{else}\ y \quad \rightsquigarrow \quad \mathtt{not} = \lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$$

Idea 1: non-commutative typing

Make the order of arguments matter: "a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should not use x after y, since $\lambda x.$ occurs before λy."

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

$$\longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y) \quad \text{which has a } y \text{ occuring before an } x$$

Caused by *2 occurrences* of z that led to *duplication* of $(x\ y)$

Idea 2: affine typing

Prohibit duplication: "a function should use its arguments *at most once*"

Variants of *linear logic* are widely used in implicit complexity!

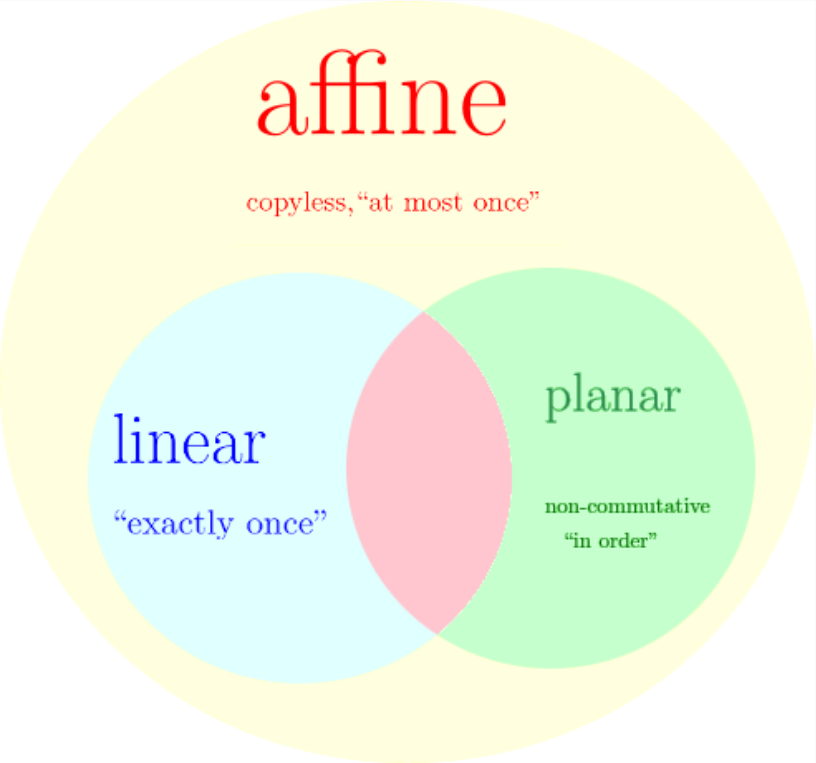
linear = "exactly once"; would work for star-free languages, but not for string functions

Substructural arrow types

$A \multimap B$: a restricted arrow type

$$A, B ::= o \mid A \to B \mid A \multimap B$$

$$\frac{\Delta \vdash t : A \multimap B \qquad \Delta' \vdash u : A}{\Delta, \Delta' \vdash t\ u : B}$$



	linear	affine	planar
$\lambda f\ x.\ f\ x\ x$	✗	✗	✗
$\lambda x\ y.\ x\ y$	✓	✓	✓
$\lambda x\ y.\ y$	✗	✓	✓
$\lambda x\ y.\ y\ x$	✓	✓	✗

Planar string languages

Church encoding with linear/affine types [Girard 1987]

$\overline{abb} = \lambda^{\rightarrow} f_a.\ \lambda^{\rightarrow} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$

- Take $\mathsf{Bool} = o \multimap o \multimap o$ in the affine setting.

Planar string languages

Church encoding with linear/affine types [Girard 1987]

$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$

- Take $\mathsf{Bool} = o \multimap o \multimap o$ in the affine setting.

Theorem (Nguyễn & P., ICALP 2020)

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

Planar string languages

Church encoding with linear/affine types [Girard 1987]

$\overline{abb} = \lambda^{\to} f_a.\ \lambda^{\to} f_b.\ \lambda^{\circ} x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to o \multimap o$

- Take $\mathsf{Bool} = o \multimap o \multimap o$ in the affine setting.

Theorem (Nguyễn & P., ICALP 2020)

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

With commutative affine types: regular languages instead

A characterization of star-free languages: proof ideas

Theorem (Nguyễn & P., ICALP 2020)

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

Key lemma for ($\Longrightarrow$)

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma\{o := A\} \multimap \mathsf{Str}_\Sigma$ with purely affine A

Proved using the Krohn–Rhodes decomposition… (linear case: more tricky)

A characterization of star-free languages: proof ideas

Theorem (Nguyễn & P., ICALP 2020)

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

Key lemma for ($\Longrightarrow$)

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma\{o := A\} \multimap \mathsf{Str}_\Sigma$ with purely affine A

Proved using the Krohn–Rhodes decomposition… (linear case: more tricky)

Key lemma for ($\Longleftarrow$)

For purely affine A, the non-commutative λ-terms of type $A \multimap A$ (modulo $=_\beta$) form a finite and aperiodic monoid.

The paper has a brute force proof; more conceptual (found afterwards):

- compile into "planar diagrams" (variant of Kauffman monoids)
  $\exists$ well-known connections: non-commutativity in linear logic $\leftrightarrow$ planarity
- characterize $\leq_{\mathcal{H}}$ on planar diagrams, deduce $\mathcal{H}$-triviality

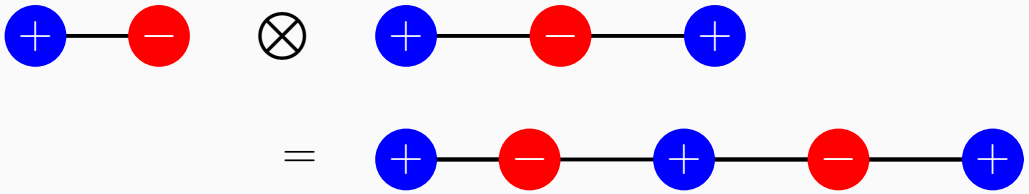
Planar diagrams (1/3): interpreting types

Interpretation of types

$[\![(o \multimap o) \multimap o]\!]$ $=$ 

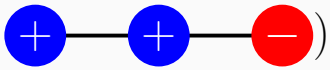
Tensorial product

put side-by-side



Duality

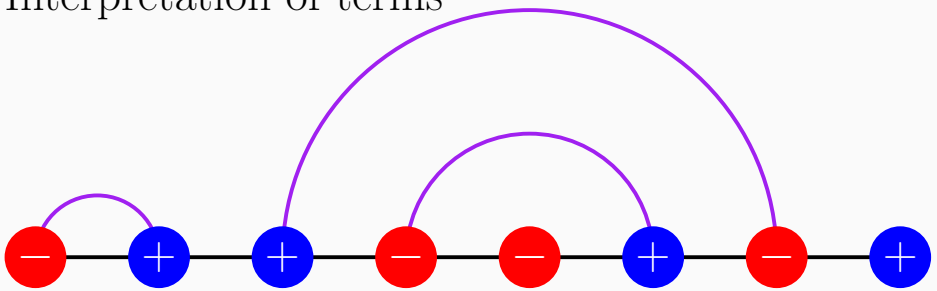
reverse the order and switch polarity

$($  $)^*$ $=$ 

$A \multimap B = A^* \otimes B$

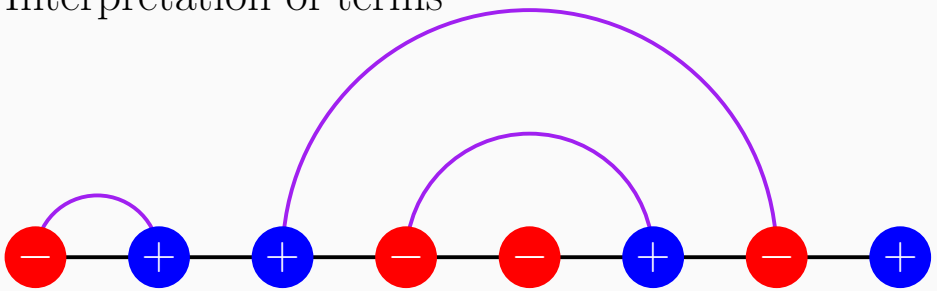
Planar diagrams (2/3): interpreting terms and composition

Interpretation of terms

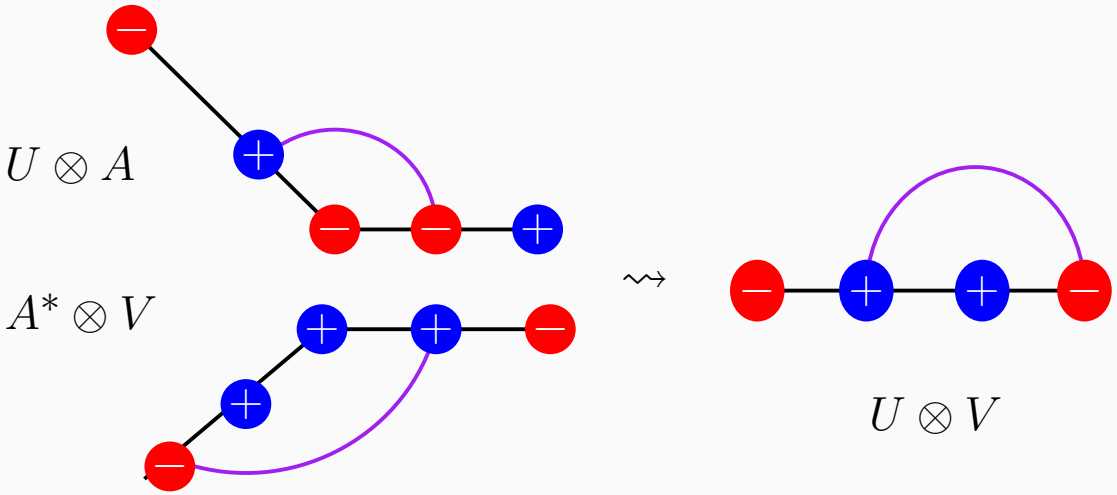


Planar diagrams (2/3): interpreting terms and composition

Interpretation of terms

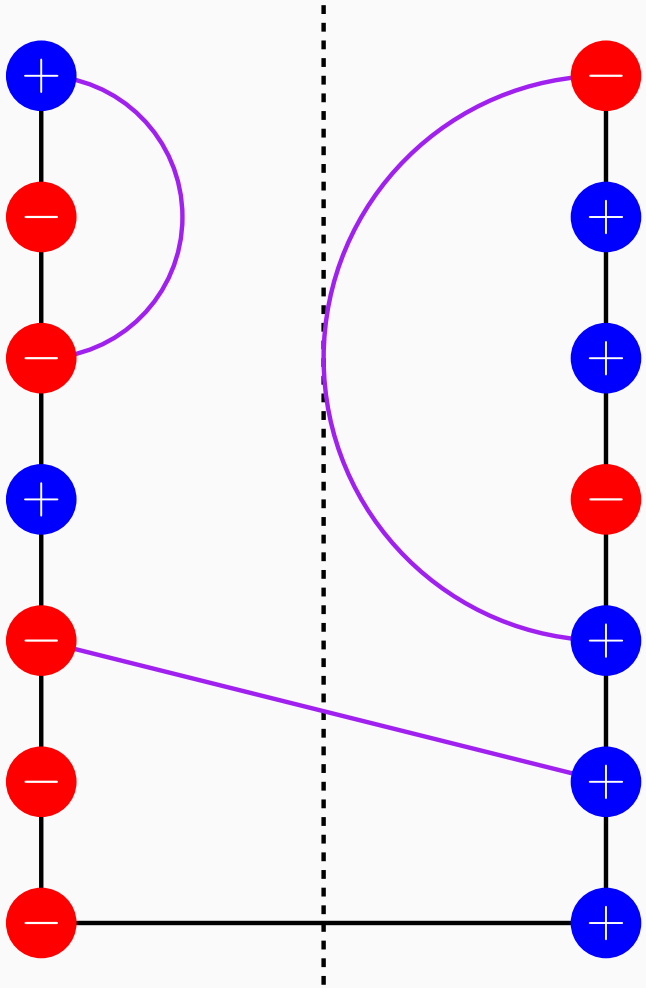


Composition



Planar diagrams (3/3): aperiodicity

Monoidal closed cat $\Rightarrow$ it suffices to show that all monoids $A \multimap A$ are aperiodic



$A^* \otimes A$

$\mathcal{R}$-class $\approx$

internal wiring on the left

$+$

positions of the input nodes

$\mathcal{L}$-class $\approx$

internal wiring on the right

$+$

position of the output nodes

planar $\Rightarrow$ $\mathcal{H}$-trivial

Back to string-to-string functions

Key lemma for star-free $\Longrightarrow$ non-commutatively λ-definable

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma\{o := A\} \multimap \mathsf{Str}_\Sigma$ with purely affine A

Does $(\supseteq)$ hold?

Back to string-to-string functions

Key lemma for star-free $\Longrightarrow$ non-commutatively λ-definable

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma\{o := A\} \multimap \mathsf{Str}_\Sigma$ with purely affine A

Does ($\supseteq$) hold? **No:** sequential fn $\subsetneq$ rational fn $\subsetneq$ **regular fn**, and

Theorem

In the commutative (*resp. non-commutative*) *affine λ-calculus,*
$\mathsf{Str}_\Gamma\{o := A\} \multimap \mathsf{Str}_\Sigma$ *with purely affine A corresponds to the* regular (*resp.* aperiodic regular) *functions.*

(also called respectively MSO transductions and FO transductions)

Back to string-to-string functions

Key lemma for star-free $\Longrightarrow$ non-commutatively λ-definable

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma\{o := A\} \multimap \mathsf{Str}_\Sigma$ with purely affine A

Does ($\supseteq$) hold? **No:** sequential fn $\subsetneq$ rational fn $\subsetneq$ **regular fn**, and

Theorem

In the commutative (*resp. non-commutative*) *affine λ-calculus,*
$\mathsf{Str}_\Gamma\{o := A\} \multimap \mathsf{Str}_\Sigma$ *with purely affine A corresponds to the* regular (*resp.* aperiodic regular) *functions.*

(also called respectively MSO transductions and FO transductions)

One possible definition for regular functions:
copyless *streaming string transducers* [Alur & Černý 2010]
$\rightsquigarrow$ 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. $\texttt{out} = XY$

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. $\texttt{out} = XY$

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. $\texttt{out} = XY$

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. $\texttt{out} = XY$

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. $\texttt{out} = XY$

Execution over *abaa*:

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

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. $\texttt{out} = XY$

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. $\texttt{out} = XY$

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. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "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. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases}$ $b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "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 $Q \times (\Sigma^*)^R$ (with $|Q| < \infty$)

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. $a \mapsto \begin{cases} X := Xa \\ Y := aY \end{cases} \quad b \mapsto \begin{cases} X := Xb \\ Y := bY \end{cases}$
- "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 $Q \times (\Sigma^*)^R$ (with $|Q| < \infty$)

Copyless SST: each register appears *at most once* on RHS of $:=$
(for each fixed input letter, at most once among all the associated $:=$)

Categorical automata theory meets denotational semantics (1)

A framework for "single-pass" automata [Colcombet & Petrişan 2017]

- internal memory = object of a *category*
- transitions = morphisms (and [letter $\mapsto$ transition] = functor)

- DFA = automata over the category of finite sets
- Copyless SSTs $\approx$ start from a category $\mathcal{R}$ of copyless register updates
  + add states by *free finite coproduct completion* $(-)_{\oplus}$

Categorical automata theory meets denotational semantics (1)

A framework for "single-pass" automata [Colcombet & Petrişan 2017]

- internal memory = object of a *category*
- transitions = morphisms (and [letter $\mapsto$ transition] = functor)

- DFA = automata over the category of finite sets
- Copyless SSTs $\approx$ start from a category $\mathcal{R}$ of copyless register updates
  + add states by *free finite coproduct completion* $(-)_{\oplus}$
- Transductions definable in affine λ-calculus can be turned into automata over a category $\mathcal{L}$ of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)

Categorical automata theory meets denotational semantics (1)

A framework for "single-pass" automata [Colcombet & Petrişan 2017]

- internal memory = object of a *category*
- transitions = morphisms (and [letter $\mapsto$ transition] = functor)

- DFA = automata over the category of finite sets
- Copyless SSTs $\approx$ start from a category $\mathcal{R}$ of copyless register updates
  + add states by *free finite coproduct completion* $(-)_\oplus$
- Transductions definable in affine λ-calculus can be turned into automata over a category $\mathcal{L}$ of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)

Proof strategy for affinely λ-definable $\Longrightarrow$ regular function

Define a *functor* $\mathcal{L} \to \mathcal{R}_\oplus$ preserving enough structure

Useful fact: there is a canonical functor from $\mathcal{L}$ to any *affine symmetric monoidal closed category* (i.e. categorical semantics of the purely affine λ-calculus)

Unfortunately $R_\oplus$ is not monoidal closed...

Categorical automata theory meets denotational semantics (2)

- $\mathcal{L}$: category of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)
- $\mathcal{R}$: category of finite sets of registers and copyless assignments
- $\mathcal{R}_\oplus$: free finite coproduct completion of the latter (add states)

We consider the free finite *product* completion: $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$

Theorem

$(\mathcal{R}_\&)_\oplus$ *is symmetric monoidal closed* (*and almost affine*).

$\Longrightarrow$ $(\mathcal{R}_\&)_\oplus$-automata are at least as powerful as $\mathcal{L}$-automata!

Categorical automata theory meets denotational semantics (2)

- $\mathcal{L}$: category of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)
- $\mathcal{R}$: category of finite sets of registers and copyless assignments
- $\mathcal{R}_\oplus$: free finite coproduct completion of the latter (add states)

We consider the free finite *product* completion: $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$

Theorem

$(\mathcal{R}_\&)_\oplus$ *is symmetric monoidal closed* (*and almost affine*).

$\Longrightarrow$ $(\mathcal{R}_\&)_\oplus$-automata are at least as powerful as $\mathcal{L}$-automata!

- $\mathcal{R}_\&$-morphisms $\approx$ single use restriction with conflicts, from streaming *tree* transducers [Alur & D'Antoni 2012]

Categorical automata theory meets denotational semantics (2)

- $\mathcal{L}$: category of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)
- $\mathcal{R}$: category of finite sets of registers and copyless assignments
- $\mathcal{R}_\oplus$: free finite coproduct completion of the latter (add states)

We consider the free finite *product* completion: $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$

Theorem

$(\mathcal{R}_\&)_\oplus$ *is symmetric monoidal closed* (*and almost affine*).

$\Longrightarrow$ $(\mathcal{R}_\&)_\oplus$-automata are at least as powerful as $\mathcal{L}$-automata!

- $\mathcal{R}_\&$-morphisms $\approx$ single use restriction with conflicts, from streaming *tree* transducers [Alur & D'Antoni 2012]
- monoidal closed categories = function space constructions available
  $\rightsquigarrow$ regular functions are closed under composition
  related to incompleteness of copyless STT wrt MSO tree transductions?

Categorical automata theory meets denotational semantics (2)

- $\mathcal{L}$: category of purely affine λ-terms (w/ const $f_c : o \multimap o$ for $c \in \Sigma$)
- $\mathcal{R}$: category of finite sets of registers and copyless assignments
- $\mathcal{R}_\oplus$: free finite coproduct completion of the latter (add states)

We consider the free finite *product* completion: $\mathcal{C}_\& = ((\mathcal{C}^{\mathrm{op}})_\oplus)^{\mathrm{op}}$

Theorem

$(\mathcal{R}_\&)_\oplus$ *is symmetric monoidal closed* (*and almost affine*).

$\Longrightarrow$ $(\mathcal{R}_\&)_\oplus$-automata are at least as powerful as $\mathcal{L}$-automata!

- $\mathcal{R}_\&$-morphisms $\approx$ single use restriction with conflicts, from streaming *tree* transducers [Alur & D'Antoni 2012]
- monoidal closed categories = function space constructions available
  $\rightsquigarrow$ regular functions are closed under composition
  related to incompleteness of copyless STT wrt MSO tree transductions?
- logic POV: connection with Dialectica interpretation / DC categories
  (coproducts of products $\approx \exists u.\, \forall x.\, \varphi(u, x)$)

Summary

λ-calculus	automata	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$	regular languages	✓[Hillebrand,Kanellakis 96]
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	regular languages	✓[Nguyễn, P. 20]
affine planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	star-free languages	✓[Nguyễn, P. 20]
linear, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathit{NC}\mathsf{Bool}$	regular languages	✓
linear planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathit{NC}\mathsf{Bool}$	star-free languages	✓

λ-calculus	transducers	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	variant of CPDA???	???
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular	✓?
planar affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	FO regular	✓?
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular functions	✓(coming soon)
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	comparison-free polyregular	??

Summary

λ-calculus	automata	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$	regular languages	✓[Hillebrand,Kanellakis 96]
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	regular languages	✓[Nguyễn, P. 20]
affine planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	star-free languages	✓[Nguyễn, P. 20]
linear, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathit{NC}\mathsf{Bool}$	regular languages	✓
linear planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathit{NC}\mathsf{Bool}$	star-free languages	✓
affine, $\mathsf{Str}_\Sigma \multimap o \to o \to o$	flip-flop monoids	✓
STLC, $\mathsf{Str}_\Sigma \to o \to o \to o$	$BC(\Sigma_1)$	??

λ-calculus	transducers	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	variant of CPDA???	???
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular	✓?
planar affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	FO regular	✓?
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular functions	✓(coming soon)
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	comparison-free polyregular	??
STLC, $\mathsf{Str}_\Sigma \to \mathsf{Str}_\Gamma$	"extended polynomials"	[Zaionc 1987]

Summary

λ-calculus	automata	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$	regular languages	✓[Hillebrand,Kanellakis 96]
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	regular languages	✓[Nguyễn, P. 20]
affine planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	star-free languages	✓[Nguyễn, P. 20]
linear, $\mathsf{Str}_\Sigma\{o := A\} \multimap NC\mathsf{Bool}$	regular languages	✓
linear planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap NC\mathsf{Bool}$	star-free languages	✓
affine, $\mathsf{Str}_\Sigma \multimap o \to o \to o$	flip-flop monoids	✓
STLC, $\mathsf{Str}_\Sigma \to o \to o \to o$	$BC(\Sigma_1)$	??

λ-calculus	transducers	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	variant of CPDA???	???
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular	✓?
planar affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	FO regular	✓?
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular functions	✓(coming soon)
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	comparison-free polyregular	??
STLC, $\mathsf{Str}_\Sigma \to \mathsf{Str}_\Gamma$	"extended polynomials"	[Zaionc 1987]

- A more principled description of the variations?
- What about trees ?

Summary

λ-calculus	automata	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$	regular languages	✓[Hillebrand,Kanellakis 96]
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	regular languages	✓[Nguyễn, P. 20]
affine planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	star-free languages	✓[Nguyễn, P. 20]
linear, $\mathsf{Str}_\Sigma\{o := A\} \multimap NC\mathsf{Bool}$	regular languages	✓
linear planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap NC\mathsf{Bool}$	star-free languages	✓
affine, $\mathsf{Str}_\Sigma \multimap o \to o \to o$	flip-flop monoids	✓
STLC, $\mathsf{Str}_\Sigma \to o \to o \to o$	$BC(\Sigma_1)$	??

λ-calculus	transducers	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	variant of CPDA???	???
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular	✓?
planar affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	FO regular	✓?
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular functions	✓(coming soon)
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	comparison-free polyregular	??
STLC, $\mathsf{Str}_\Sigma \to \mathsf{Str}_\Gamma$	"extended polynomials"	[Zaionc 1987]

- A more principled description of the variations?
- What about trees ?

Summary

λ-calculus	automata	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Bool}$	regular languages	✓[Hillebrand,Kanellakis 96]
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	regular languages	✓[Nguyễn, P. 20]
affine planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Bool}$	star-free languages	✓[Nguyễn, P. 20]
linear, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathit{NC}\mathsf{Bool}$	regular languages	✓
linear planar, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathit{NC}\mathsf{Bool}$	star-free languages	✓
affine, $\mathsf{Str}_\Sigma \multimap o \to o \to o$	flip-flop monoids	✓
STLC, $\mathsf{Str}_\Sigma \to o \to o \to o$	$BC(\Sigma_1)$	??

λ-calculus	transducers	status
STLC, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	variant of CPDA???	???
affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular	✓?
planar affine, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	FO regular	✓?
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \multimap \mathsf{Str}_\Gamma$	regular functions	✓(coming soon)
linear with additives, $\mathsf{Str}_\Sigma\{o := A\} \to \mathsf{Str}_\Gamma$	comparison-free polyregular	??
STLC, $\mathsf{Str}_\Sigma \to \mathsf{Str}_\Gamma$	"extended polynomials"	[Zaionc 1987]

- A more principled description of the variations?
- What about trees ?

Thanks for your attention! Questions?