Star-free languages, first-order transductions and the non-commutative λ-calculus

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

IRIF automata seminar, March 19th, 2021

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.
Many theoretical PLs *not* Turing-complete, especially *typed* λ*-calculi*

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.
Many theoretical PLs *not* Turing-complete, especially *typed* λ*-calculi*
Implicit complexity: machine-free characterizations of complexity classes
using high-level programming languages
Big project: the same thing for automata instead of complexity

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.
Many theoretical PLs *not* Turing-complete, especially *typed* λ*-calculi*
Implicit complexity: machine-free characterizations of complexity classes
using high-level programming languages
Big project: the same thing for automata instead of complexity

Plan

1. How to compute string functions with the simply-typed λ-calculus
2. A correspondence between star-free languages and the non-commutative λ-calculus
3. Discussion of a semantic evaluation proof and further work on FO transductions

The λ-calculus

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

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

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

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

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 $\longrightarrow_\beta$

The λ-calculus

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

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

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

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

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

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

Simply typed functions on Church numerals (1)

Church encodings of (unary) natural numbers:

- $\mathsf{Nat} = (o \to o) \to o \to o$
- $n \in \mathbb{N} \rightsquigarrow \overline{n} = \lambda f.\, \lambda x.\, f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat}$ with n times f
- all inhabitants of Nat are equal to some $\overline{n}$ up to $=_{\beta\eta}$

Theorem (Schwichtenberg 1975)

The functions $\mathbb{N} \to \mathbb{N}$ definable by simply-typed λ-terms of type $\mathsf{Nat} \to \mathsf{Nat}$ are the extended polynomials (generated by 0, 1, $+$, $\times$, $\mathtt{id}$ and $\mathtt{ifzero}$).

Simply typed functions on Church numerals (1)

Church encodings of (unary) natural numbers:

- $\mathsf{Nat} = (o \to o) \to o \to o$
- $n \in \mathbb{N} \rightsquigarrow \overline{n} = \lambda f.\, \lambda x.\, f\,(\ldots\ (f\,x)\ldots) : \mathsf{Nat}$ with n times f
- all inhabitants of Nat are equal to some $\overline{n}$ up to $=_{\beta\eta}$

Theorem (Schwichtenberg 1975)

The functions $\mathbb{N} \to \mathbb{N}$ definable by simply-typed λ-terms of type $\mathsf{Nat} \to \mathsf{Nat}$ are the extended polynomials (generated by 0, 1, $+$, $\times$, $\mathtt{id}$ and $\mathtt{ifzero}$).

Let's add a bit of (meta-level) polymorphism: $t = \mathsf{Nat}[A] \to \mathsf{Nat}$
where $\mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$

Open question

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

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\,(\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated!

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\,(\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated! Smaller types, still heterogenous:

$$\mathsf{exp2} = \lambda n.\, n\ \overline{2} : \mathsf{Nat}[o \to o] \to \mathsf{Nat}$$

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\ (\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated! Smaller types, still heterogenous:

$$\mathsf{exp2} = \lambda n.\, n\ \overline{2} : \mathsf{Nat}[o \to o] \to \mathsf{Nat}$$

Towers of exponentials of *any fixed height* $\mathsf{Nat}[T[A]] \to \mathsf{Nat}[A]$.
This is the fastest possible growth for simply typed λ-terms.

Simply typed functions on Church numerals (2)

Take $\mathsf{mult} = \lambda n.\lambda m.\lambda f.\, n\,(m\,f) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.
$\mathsf{mult}\ \overline{2} : \mathsf{Nat} \to \mathsf{Nat}$ can be iterated by a $\mathsf{Nat}[\mathsf{Nat}]$...

$$\longrightarrow \mathsf{exp2} = \lambda n.\, n\ (\mathsf{mult}\ \overline{2})\ \overline{1} : \mathsf{Nat}[\mathsf{Nat}] \to \mathsf{Nat}$$

which cannot be iterated! Smaller types, still heterogenous:

$$\mathsf{exp2} = \lambda n.\, n\ \overline{2} : \mathsf{Nat}[o \to o] \to \mathsf{Nat}$$

Towers of exponentials of *any fixed height* $\mathsf{Nat}[T[A]] \to \mathsf{Nat}[A]$.
This is the fastest possible growth for simply typed λ-terms.

On the other hand:

Theorem (Statman 198X)

Subtraction cannot be defined by any simply typed $t : \mathsf{Nat}[A] \to \mathsf{Nat}[B] \to \mathsf{Nat}$.

Simply typed functions on Church numerals (3)

Open question

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

Subtraction cannot be defined; some "easy" 1-variable functions, e.g. $n \mapsto \lfloor \sqrt{n} \rfloor$, are also undefinable.

Does this even admit a *satisfying* answer?

Simply typed functions on Church numerals (3)

Open question

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

Subtraction cannot be defined; some "easy" 1-variable functions,
e.g. $n \mapsto \lfloor \sqrt{n} \rfloor$, are also undefinable.

Does this even admit a *satisfying* answer? There is one for *predicates*!

Theorem (Joly 2001)

A subset of $\mathbb{N}^k$ *is decidable by some* $t : \mathsf{Nat}[A_1] \to \cdots \to \mathsf{Nat}[A_n] \to \mathsf{Bool}$
(*where* $\mathsf{Bool} = o \to o \to o$) *if and only if it is ultimately periodic.*

Simply typed functions on Church numerals (3)

Open question

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

Subtraction cannot be defined; some "easy" 1-variable functions,
e.g. $n \mapsto \lfloor \sqrt{n} \rfloor$, are also undefinable.

Does this even admit a *satisfying* answer? There is one for *predicates*!

Theorem (Joly 2001)

A subset of $\mathbb{N}^k$ *is decidable by some* $t : \mathsf{Nat}[A_1] \to \cdots \to \mathsf{Nat}[A_n] \to \mathsf{Bool}$
(*where* $\mathsf{Bool} = o \to o \to o$) *if and only if it is ultimately periodic.*

Corollary

If $t : \mathsf{Nat}[A] \to \mathsf{Nat}$ *defines* $f_t : \mathbb{N} \to \mathbb{N}$, *then*
$X \subseteq \mathbb{N}$ *ultimately periodic* $\implies f_t^{-1}(X)$ *ultimately periodic.*

A not quite trivial necessary condition!

Simply typed functions on Church-encoded strings

To gain more insight, let's *generalize*! $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$

Church encodings of *strings* over alphabet $\Sigma = \{a, b\}$:

- $\mathsf{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$
- $abb \in \{a, b\}^* \rightsquigarrow \overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_\Sigma$

More generally $\mathsf{Str}_\Sigma = (o \to o) \to \ldots |\Sigma| \text{ times} \ldots \to (o \to o) \to o \to o$

Open question

Choose some simple type A and some term $t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$.
What functions $\Gamma^* \to \Sigma^*$ can be defined this way?

Without input type substitutions, an answer is known [Zaionc 1987].

Simply typed functions on Church-encoded strings

To gain more insight, let's *generalize*! $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$

Church encodings of *strings* over alphabet $\Sigma = \{a, b\}$:

- $\mathsf{Str}_{\{a,b\}} = (o \to o) \to (o \to o) \to o \to o$
- $abb \in \{a, b\}^* \rightsquigarrow \overline{abb} = \lambda f_a.\ \lambda f_b.\ \lambda x.\ f_a\ (f_b\ (f_b\ x)) : \mathsf{Str}_\Sigma$

More generally $\mathsf{Str}_\Sigma = (o \to o) \to \ldots |\Sigma| \text{ times} \ldots \to (o \to o) \to o \to o$

Open question

Choose some simple type A and some term $t : \mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$.
What functions $\Gamma^* \to \Sigma^*$ can be defined this way?

Without input type substitutions, an answer is known [Zaionc 1987].

An answer for predicates [Hillebrand & Kanellakis 1996]

A subset of Σ^* is decidable by some $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$
if and only if it is a *regular language*.

Note: unary regular languages $\cong$ ultimately periodic subsets of $\mathbb{N}$

λ-definable functions are regular

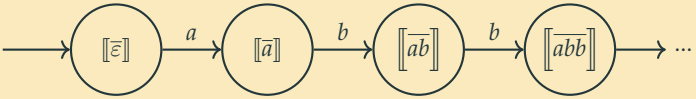
Theorem (Hillebrand & Kanellakis, LICS'96)

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

Proof by semantic evaluation.

Let $[\![-]\!]$ stand for the denotational semantics in the *CCC of finite sets*.

We build an automaton with *finite* set of states $Q = [\![\mathsf{Str}_\Sigma[A]]\!]$ ($\mathrm{Card}(Q)$ depends on A), acceptation as $[\![t]\!](-) = [\![\mathtt{true}]\!]$.



$$t\,\overline{w} =_\beta \mathtt{true} \iff [\![t]\!]([\![\overline{w}]\!]) = [\![\mathtt{true}]\!] \iff w \text{ accepted}$$

(Proof of ($\Leftarrow$): if $\mathrm{Card}([\![o]\!]) \geq 2$ then $[\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]$) □

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

Reminder: star-free languages and aperiodic monoids

Our goal now: characterize *star-free languages*.

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*.

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*.

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 $P \subseteq M$.

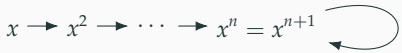
Reminder: star-free languages and aperiodic monoids

Our goal now: characterize *star-free languages*.

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*.

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*.

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? $\texttt{not}^{n+1} \neq \texttt{not}^{n}$

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

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

From aperiodicity to non-commutative affine types

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

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

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

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? $\texttt{not}^{n+1} \neq \texttt{not}^{n}$

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

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

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? $\texttt{not}^{n+1} \neq \texttt{not}^{n}$

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

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

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? $\texttt{not}^{n+1} \neq \texttt{not}^{n}$

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

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

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? $\texttt{not}^{n+1} \neq \texttt{not}^{n}$

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

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

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!

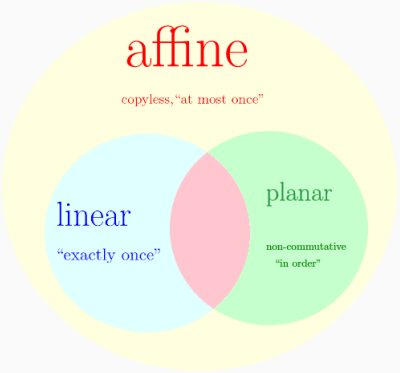
linear = "exactly once"; would work for star-free languages, but not for string functions

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
$\lambda f\, x.\, f\, x\, x$	✗	✗	✗
$\lambda x\, y.\, x\, y$	✓	✓	✓
$\lambda x\, y.\, y$	✗	✓	✓
$\lambda x\, y.\, y\, x$	✓	✓	✗

Formally

Typing judgments: {non-affine variables} | {affine variables} $\vdash t : A$

- $\lambda^{\rightarrow} x.\, t : A \to B$ unrestricted function
- $\lambda^{\circ} x.\, t : A \multimap B$ affine function (at most one x in t)

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

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

The above = Dual Intuitionistic Linear Logic

$$\frac{\Gamma \mid \Delta \vdash t : A}{\Gamma \mid \Delta' \vdash t : A} \text{ when } \Delta \subseteq \Delta' \text{: } \textit{weakening} \text{ rule}$$

Linear types allow for more precise characterizations of Church encodings.

Church encoding with linear/affine types [Girard 1987]

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

Linear types allow for more precise characterizations of Church encodings.

Church encoding with linear/affine types [Girard 1987]

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

Main theorem

$L \subseteq \Sigma^*$ is *star-free* $\iff$ L is defined by some $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Bool}$ with A *purely affine*, i.e. containing no '$\to$'

- The result also holds with strictly linear types
- If $\otimes$ is made commutative, we obtain regular languages instead

A characterization of star-free languages: proof ideas

Main theorem

$L \subseteq \Sigma^*$ is *star-free* $\iff$ L is defined by some $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Bool}$ with A *purely affine*, i.e. containing no '$\to$'

Key lemma for ($\Longrightarrow$)

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ with purely affine A

This is proved using the Krohn–Rhodes theorem

A characterization of star-free languages: proof ideas

Main theorem

$L \subseteq \Sigma^*$ is *star-free* $\iff$ L is defined by some $t : \mathsf{Str}_\Sigma[A] \multimap \mathtt{Bool}$ with A *purely affine*, i.e. containing no '$\to$'

Key lemma for ($\Longrightarrow$)

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ with purely affine A

This is proved using the Krohn–Rhodes theorem

Key lemma 1 for ($\Longleftarrow$): normal form for λ-definable functions

Every λ-term $\mathsf{Str}_\Gamma[A] \multimap \mathtt{Bool}$ is equivalent to one of the form $\lambda s.\ o\ (s\ d_1 \dots\ d_{|\Gamma|})$

- Allows to reduce to studying purely affine terms $o, d_1, \dots, d_{|\Gamma|}$

Key lemma 2 for ($\Longleftarrow$): aperiodicity

Non-commutative λ-terms of type $A \multimap A$ (modulo $=_\beta$) form a finite aperiodic monoid.

- This is proven syntactically, by induction over A in the paper

A characterization of star-free languages: proof ideas

Main theorem

$L \subseteq \Sigma^*$ is *star-free* $\iff$ L is defined by some $t : \mathsf{Str}_\Sigma[A] \multimap \mathtt{Bool}$ with A *purely affine*, i.e. containing no '$\to$'

Key lemma for ($\Longrightarrow$)

aperiodic sequential functions $\subseteq \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ with purely affine A

This is proved using the Krohn–Rhodes theorem

Key lemma 1 for ($\Longleftarrow$): normal form for λ-definable functions

Every λ-term $\mathsf{Str}_\Gamma[A] \multimap \mathtt{Bool}$ is equivalent to one of the form $\lambda s.\ o\ (s\ d_1 \dots\ d_{|\Gamma|})$

- Allows to reduce to studying purely affine terms $o, d_1, \dots, d_{|\Gamma|}$

Key lemma 2 for ($\Longleftarrow$): aperiodicity

Non-commutative λ-terms of type $A \multimap A$ (modulo $=_\beta$) form a finite aperiodic monoid.

- This is proven syntactically, by induction over A in the paper
- **Now:** an alternative proof sketch replacing 2 by semantic evaluation

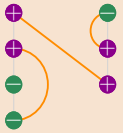
A category of planar diagrams

- Interpret purely linear non-commutative λ-terms in a monoidal closed category
- We consider a non-commutative refinement of Geometry of Interaction

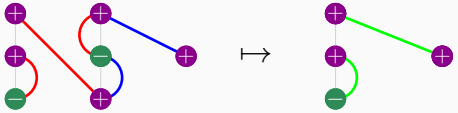
(well-known model of linear logic)

A compact closed category of planar diagrams

- **Objects:** words in $\{+, -\}^*$
- **Morphisms** $u \to v$ **:** graphs over $|u| + |v|$ with
  - degree ≤ 1 for every node
  - polarity restrictions
  - planarity restriction



To compute the composition of two morphisms, follow the paths (and forget the middle component)

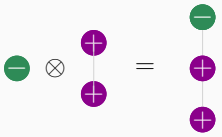


Compact-closure and interpretation of the λ-calculus

Structure to interpret the linear λ-calculus

- Monoidal product $A \otimes B$ given by concatenation
- Duals A^*: reverse and flip polarities
- Monoidal closure by setting $A \multimap B \;=\; A^* \otimes B$
- Interpretation of types $[\![A]\!]$ by induction with $[\![o]\!] = +$

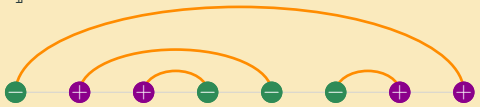
  (injective interpretation of booleans)



Examples

$[\![(o \multimap o) \multimap o \multimap o) \multimap ((o \multimap o) \multimap o) \multimap o]\!] \;=\; - + + - - - + +$

$[\![\lambda f.\lambda g.\, f\ (\lambda x.\ x)\ (g\ (\lambda x.\ x))]\!] \;=$



Aperiodicity

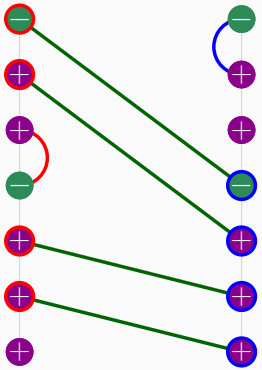
To conclude, we need to show that every $(\mathtt{Hom}(A, A), \circ)$ is finite and aperiodic for every A

$\mathcal{R}$-class determined by:

the internal wiring on the left

+

positions of the input nodes



$\mathcal{L}$-class determined by:

the internal wiring on the left

+

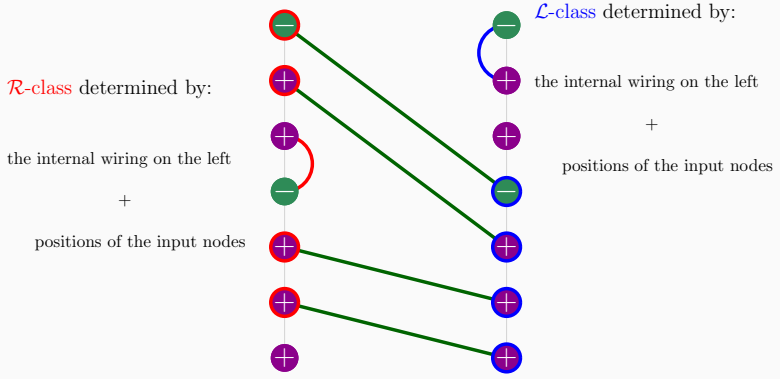
positions of the input nodes

Therefore: planar $\implies$ $\mathcal{H}$-trivial

- More elementary proofs w/o Green relations possible (e.g. order+Kleene's theorem)

Aperiodicity

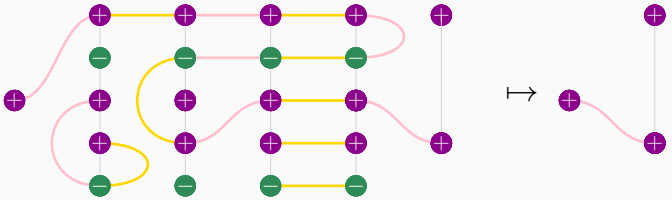
To conclude, we need to show that every $(\mathtt{Hom}(A,A),\circ)$ is finite and aperiodic for every A



Therefore: planar $\Longrightarrow$ $\mathcal{H}$-trivial

- More elementary proofs w/o Green relations possible (e.g. order+Kleene's theorem)
- Planarity restriction is essential (consider )

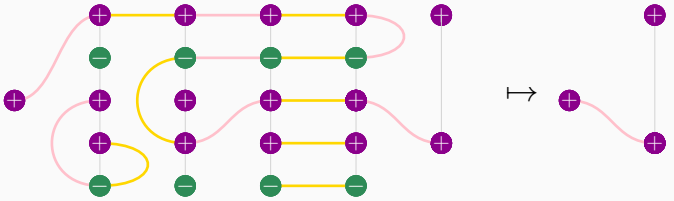
Non-planar diagrams (with crossings): reminiscent of runs in 2DFAs!



- Transition functions $\delta : \Sigma \to \mathsf{Hom}(Q, Q)$ for some object Q $\quad Q \approx$ set of directed states
- (actually, should also incorporate boundary morphisms $\mathsf{Hom}(+, Q)$ and $\mathsf{Hom}(Q, F)$)

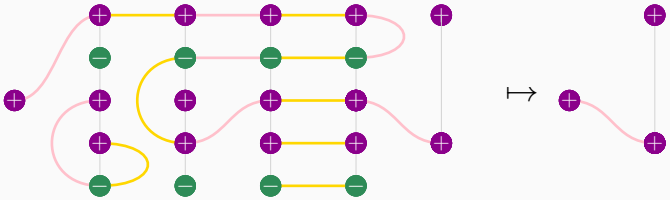
Diagrams and two-way automata

Non-planar diagrams (with crossings): reminiscent of runs in 2DFAs!



- Transition functions $\delta : \Sigma \to \mathsf{Hom}(Q, Q)$ for some object Q $\quad Q \approx$ set of directed states
- (actually, should also incorporate boundary morphisms $\mathsf{Hom}(+, Q)$ and $\mathsf{Hom}(Q, F)$)
- Planarity restriction $\Rightarrow$ the transition flow monoid is aperiodic
- (links between GoI and planar 2DFAs already considered by (Hines 2003))

Non-planar diagrams (with crossings): reminiscent of runs in 2DFAs!



- Transition functions $\delta : \Sigma \to \mathsf{Hom}(Q, Q)$ for some object Q $\quad Q \approx$ set of directed states
- (actually, should also incorporate boundary morphisms $\mathsf{Hom}(+, Q)$ and $\mathsf{Hom}(Q, F)$)
- Planarity restriction $\Rightarrow$ the transition flow monoid is aperiodic
- (links between GoI and planar 2DFAs already considered by (Hines 2003))

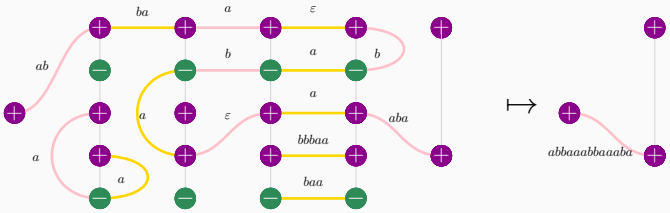
Theorem

Star-free languages are exactly those recognized by planar 2DFAs.

More generally: first-order transductions

Consider a richer category of diagrams where edges are labelled by output words

(labels of compositions given by concatenation)



Much like before, corresponding notion of (planar) 2DFTs.

Theorem

First-order transduction (FO regular functions) are exactly those computed by reversible planar 2DFTs.

- 2DFTs with aperiodic transition monoid = FO regular functions [Carton&Dartois 2015]

  (hence reversible planar 2DFTs $\subseteq$ FO-transductions)

- FO transduction $\subseteq$ reversible planar 2DFTs: closure by composition and Krohn–Rhodes

  (see `http://nguyentito.eu/2021-01-links.pdf`)

Back to string-to-string functions in the λ-calculus

Having a compact-closed category of diagrams ensures the following.

Consequence of the above

Every function implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative linear λ-calculus with A purely linear is a FO-transduction.

Back to string-to-string functions in the λ-calculus

Having a compact-closed category of diagrams ensures the following.

Consequence of the above

Every function implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative linear λ-calculus with A purely linear is a FO-transduction.

However they are much weaker than FO-transductions

Preservation of Parikh image in the linear case

For every function $f : \Sigma^* \to \Gamma^*$ implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative **linear** λ-calculus, there is a function $\widehat{f} : \mathbb{N}^\Sigma \to \mathbb{N}^\Gamma$ such that $|f(w)|_b = \widehat{f}(a \mapsto |w|_a)$.

- Proof by semantic evaluation: commutative monoid = one-object compact-closed category

(use the monoid $(\mathbb{N}, 0, +)$ lifted pointwise)

Back to string-to-string functions in the λ-calculus

Having a compact-closed category of diagrams ensures the following.

Consequence of the above

Every function implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative linear λ-calculus with A purely linear is a FO-transduction.

However they are much weaker than FO-transductions

Preservation of Parikh image in the linear case

For every function $f : \Sigma^* \to \Gamma^*$ implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative **linear** λ-calculus, there is a function $\widehat{f} : \mathbb{N}^\Sigma \to \mathbb{N}^\Gamma$ such that $|f(w)|_b = \widehat{f}(a \mapsto |w|_a)$.

- Proof by semantic evaluation: commutative monoid = one-object compact-closed category

  (use the monoid $(\mathbb{N}, 0, +)$ lifted pointwise)

- Not true anymore if we consider the **affine** non-commutative λ-calculus

  (no terminal objects in our categories)

Back to string-to-string functions in the λ-calculus

Having a compact-closed category of diagrams ensures the following.

Consequence of the above

Every function implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative linear λ-calculus with A purely linear is a FO-transduction.

However they are much weaker than FO-transductions

Preservation of Parikh image in the linear case

For every function $f : \Sigma^* \to \Gamma^*$ implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative **linear** λ-calculus, there is a function $\widehat{f} : \mathbb{N}^\Sigma \to \mathbb{N}^\Gamma$ such that $|f(w)|_b = \widehat{f}(a \mapsto |w|_a)$.

- Proof by semantic evaluation: commutative monoid = one-object compact-closed category

  (use the monoid $(\mathbb{N}, 0, +)$ lifted pointwise)

- Not true anymore if we consider the **affine** non-commutative λ-calculus

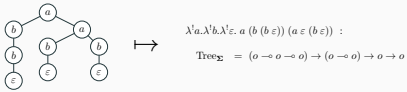
  (no terminal objects in our categories)

Working conjecture

Every function implemented by a term $t : \mathsf{Str}_\Sigma[A] \multimap \mathsf{Str}_\Gamma$ in the non-commutative affine λ-calculus with A purely affine is a FO-transduction and **vice-versa**.

Some remarks about trees

- Ranked trees can be handled using Church encodings, e.g., for $Sigma = \{a : 2, b : 1, \varepsilon : 0\}$



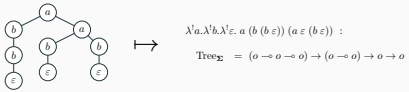
Characterization of tree transductions in linear logic with additives

regular *tree* function $\iff$ definable by some $t : \mathtt{Tree}_\Gamma[A] \multimap \mathtt{Tree}_\Sigma$ in linear λ-calculus *with additives* and A purely linear

- In that result A can feature sums $\oplus$ and cartesian products $\&$
- Completeness does not appeal to Krohn-Rhodes (intuition: states Q can be modeled by $1 \oplus \ldots \oplus 1$)

Some remarks about trees

- Ranked trees can be handled using Church encodings, e.g., for $Sigma = \{a : 2, b : 1, \varepsilon : 0\}$



Characterization of tree transductions in linear logic with additives

regular *tree* function $\iff$ definable by some $t : \mathtt{Tree}_\Gamma[A] \multimap \mathtt{Tree}_\Sigma$ in linear λ-calculus *with additives* and A purely linear

- In that result A can feature sums $\oplus$ and cartesian products $\&$
- Completeness does not appeal to Krohn-Rhodes (intuition: states Q can be modeled by $1 \oplus \ldots \oplus 1$)
- What happens if we remove additives connectives?

Some remarks about trees

- Ranked trees can be handled using Church encodings, e.g., for $Sigma = \{a : 2, b : 1, \varepsilon : 0\}$

$\mapsto$

$\lambda^! a.\lambda^! b.\lambda^! \varepsilon.\ a\ (b\ (b\ \varepsilon))\ (a\ \varepsilon\ (b\ \varepsilon))\ :$

$\mathsf{Tree}_\Sigma\ =\ (o \multimap o \multimap o) \to (o \multimap o) \to o \to o$

Characterization of tree transductions in linear logic with additives

regular *tree* function $\iff$ definable by some $t : \mathsf{Tree}_\Gamma[A] \multimap \mathsf{Tree}_\Sigma$ in linear λ-calculus *with additives* and A purely linear

- In that result A can feature sums $\oplus$ and cartesian products $\&$
- Completeness does not appeal to Krohn-Rhodes (intuition: states Q can be modeled by $1 \oplus \ldots \oplus 1$)
- What happens if we remove additives connectives?

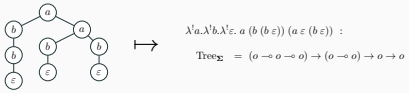
Additive connectives are required for trees!

Languages $L \subseteq \mathbf{Tree}(\Sigma)$ recognized by terms $\mathsf{Tree}_\Sigma[A] \multimap \mathrm{Bool}$ of the commutative linear λ-calculus with A purely linear are recognizabe by tree-walking automata.

- Via semantic evaluation using our category of diagrams
- Tree-walking automata do not recognize every regular language (Colcombet&Bojańczyk 2008)

Some remarks about trees

- Ranked trees can be handled using Church encodings, e.g., for $Sigma = \{a : 2, b : 1, \varepsilon : 0\}$



Characterization of tree transductions in linear logic with additives

regular *tree* function $\iff$ definable by some $t : \mathtt{Tree}_\Gamma[A] \multimap \mathtt{Tree}_\Sigma$ in linear λ-calculus *with additives* and A purely linear

- In that result A can feature sums $\oplus$ and cartesian products $\&$
- Completeness does not appeal to Krohn-Rhodes (intuition: states Q can be modeled by $1 \oplus \ldots \oplus 1$)
- What happens if we remove additives connectives?

Additive connectives are required for trees!

Languages $L \subseteq \mathbf{Tree}(\Sigma)$ recognized by terms $\mathtt{Tree}_\Sigma[A] \multimap \mathrm{Bool}$ of the commutative linear λ-calculus with A purely linear are recognizabe by tree-walking automata.

- Via semantic evaluation using our category of diagrams
- Tree-walking automata do not recognize every regular language (Colcombet&Bojańczyk 2008)
- We suspect that there is a correspondance in the affine case

Conclusion

Today:

- Church encodings lead to connections with automata
- A characterization of star-free languages
- Ideas towards a characterization of FO transductions via planarity

Broader picture

$\mathsf{Str}_\Sigma[A] \multimap \mathtt{Bool}$ with A linear (adapted as needed):

λ-calculus	languages	status
simply typed	regular	✓ [Hillebrand & Kanellakis 1996]
linear or affine	regular	✓
non-commutative linear or affine	star-free	✓

$\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ with A affine (adapted as needed):

λ-calculus	transducers	status
affine	regular functions	✓
non-commutative affine	first-order regular fn.	✓?
linear/affine with additives	regular functions	✓
parsimonious	polyregular	??
simply typed	variant of CPDA???	???

Conclusion

Today:

- Church encodings lead to connections with automata
- A characterization of star-free languages
- Ideas towards a characterization of FO transductions via planarity

Broader picture

$\mathsf{Str}_\Sigma[A] \multimap \mathtt{Bool}$ with A linear (adapted as needed):

λ-calculus	languages	status
simply typed	regular	✓ [Hillebrand & Kanellakis 1996]
linear or affine	regular	✓
non-commutative linear or affine	star-free	✓

$\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ with A affine (adapted as needed):

λ-calculus	transducers	status
affine	regular functions	✓
non-commutative affine	first-order regular fn.	✓?
linear/affine with additives	regular functions	✓
parsimonious	polyregular	??
simply typed	variant of CPDA???	???

+ a characterization of $\mathsf{Str}[A] \to \mathsf{Str}$ **as comparison-free polyregular functions**

Conclusion

Today:

- Church encodings lead to connections with automata
- A characterization of star-free languages
- Ideas towards a characterization of FO transductions via planarity

Broader picture

$\mathsf{Str}_\Sigma[A] \multimap \mathtt{Bool}$ with A linear (adapted as needed):

λ-calculus	languages	status
simply typed	regular	✓ [Hillebrand & Kanellakis 1996]
linear or affine	regular	✓
non-commutative linear or affine	star-free	✓

$\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ with A affine (adapted as needed):

λ-calculus	transducers	status
affine	regular functions	✓
non-commutative affine	first-order regular fn.	✓?
linear/affine with additives	regular functions	✓
parsimonious	polyregular	??
simply typed	variant of CPDA???	???

+ a characterization of $\mathsf{Str}[A] \to \mathsf{Str}$ **as comparison-free polyregular functions**

Thanks for listening!