Implicit automata in typed λ-calculi

Lê Thành Dũng (Tito) Nguyễn — `https://nguyentito.eu` — IRISA, Rennes
joint work with Pierre Pradic (Swansea University)

Journées du GT Scalp, 3 novembre 2021

There is a remarkable divide in the field of logic in Computer Science, between two distinct strands: one focusing on semantics and compositionality ("Structure"), the other on expressiveness and complexity ("Power"). It is remarkable because these two fundamental aspects of our field are studied using almost disjoint technical languages and methods, by almost disjoint research communities.

— From the blurb of this summer's *Structure meets Power* workshop
(also found in Abramsky & Shah CSL'18)

There is a remarkable divide in the field of logic in Computer Science, between two distinct strands: one focusing on semantics and compositionality ("Structure"), the other on expressiveness and complexity ("Power"). It is remarkable because these two fundamental aspects of our field are studied using almost disjoint technical languages and methods, by almost disjoint research communities.

— From the blurb of this summer's *Structure meets Power* workshop
(also found in Abramsky & Shah CSL'18)

- The Scalp community cares about "structure"
- This talk: connections with automata, from the "power" side
  - Are they really though? I'll come back to that at the end

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete
$\longrightarrow$ typical "power" question: *what* does it compute?

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete $\longrightarrow$ typical "power" question: *what* does it compute? Some results known, e.g.

Theorem (Schwichtenberg 1975)

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

where Nat is the type of *Church numerals*.

Some motivations coming from the λ-calculus

Let's consider the *simply typed λ-calculus* (I assume basic familiarity).

It's a programming language, so it computes, right? And it's not Turing-complete $\longrightarrow$ typical "power" question: *what* does it compute? Some results known, e.g.

Theorem (Schwichtenberg 1975)

The functions $\mathbb{N}^k \to \mathbb{N}$ definable by simply-typed λ-terms $t : \mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ are the extended polynomials (generated by 0, 1, $+$, $\times$, `id` and `ifzero`).

where Nat is the type of *Church numerals*.

But some open questions have no known satisfying answer...

Simply typed functions on Church numerals (1)

Recall that the type of Church numerals is $\mathsf{Nat} = (o \to o) \to o \to o$

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

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

Schwichtenberg 1975: $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = extended polynomials

Simply typed functions on Church numerals (1)

Recall that the type of Church numerals is $\mathsf{Nat} = (o \to o) \to o \to o$

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

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

Schwichtenberg 1975: $\mathsf{Nat} \to \cdots \to \mathsf{Nat} \to \mathsf{Nat}$ = extended polynomials

Let's add a bit of (meta-level) polymorphism: for $n \in \mathbb{N}$,

$$\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$$

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)

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? (where $B[A] = B\{o := A\}$)

Why is nobody working on this seemingly natural question?

- Apparently, low hopes for a nice answer until now
  - you can express towers of exponentials
  - but not subtraction or equality (Statman 198X)
- Not so important: this is about "power" while our focus is on "structure"

Simply typed functions on Church numerals (2)

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? (where $B[A] = B\{o := A\}$)

Why is nobody working on this seemingly natural question?

- Apparently, low hopes for a nice answer until now
  - you can express towers of exponentials
  - but not subtraction or equality (Statman 198X)
- Not so important: this is about "power" while our focus is on "structure"

Little-known(?) fact: the case $\mathbb{N} \to \{0, 1\}$ / $\mathsf{Nat}[A] \to \mathsf{Bool}$ has a very satisfying characterization, that even generalizes to strings!

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{\mathtt{1}\}}$):

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

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

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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

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

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

Defining languages in the simply typed λ-calculus

Church encodings of binary strings [Böhm & Berarducci 1985]

$\simeq$ `fold_right` on a list of characters (generalizable to any alphabet; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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

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

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

Theorem (Hillebrand & Kanellakis 1996)

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

Regular languages in STλC and implicit complexity

Template for theorems at the structure/power interface

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

- Hillebrand & Kanellakis: $\mathcal{P}$ = simply typed λ-calculus, $\mathcal{C}$ = regular languages
  - Good news: unlike "extended polynomials", a central object in another field of computer science, namely *automata theory*
  - The definition will be recalled soon

Regular languages in STλC and implicit complexity

Template for theorems at the structure/power interface

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

- Hillebrand & Kanellakis: $\mathcal{P}$ = simply typed λ-calculus, $\mathcal{C}$ = regular languages
  - Good news: unlike "extended polynomials", a central object in another field of computer science, namely *automata theory*
  - The definition will be recalled soon
- *Implicit computational complexity*: $\mathcal{C}$ is a complexity class e.g. P, NP, ...
  - ICC has been an active research field since the 1990s (cf. Péchoux's HDR)
  - Historical example (Girard): $\mathcal{P}$ = Light Linear Logic, $\mathcal{C} = \mathsf{P}$ (polynomial time)

Regular languages in STλC and implicit complexity

Template for theorems at the structure/power interface

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

- Hillebrand & Kanellakis: $\mathcal{P}$ = simply typed λ-calculus, $\mathcal{C}$ = regular languages
  - Good news: unlike "extended polynomials", a central object in another field of computer science, namely *automata theory*
  - The definition will be recalled soon
- *Implicit computational complexity*: $\mathcal{C}$ is a complexity class e.g. P, NP, ...
  - ICC has been an active research field since the 1990s (cf. Péchoux's HDR)
  - Historical example (Girard): $\mathcal{P}$ = Light Linear Logic, $\mathcal{C} = \mathsf{P}$ (polynomial time)

Our "implicit automata" research programme: $\mathcal{C}$ coming from automata theory

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

Grandeur et misère de la complexité implicite

Implicit complexity has been very successful in capturing lots of different complexity classes! But the programming languages involved are often ad-hoc...

> *Several systems* [...] *have been produced; my favourite being* **LLL**, *light linear logic, which* [...] *can harbour all polytime functions. Unfortunately these systems are good for nothing, they all come from bondage: artificial restrictions on the rules which achieve certain effects, but are not justified by use, not even by some natural "semantic" considerations.* — Girard, *From Foundations to Ludics*

Grandeur et misère de la complexité implicite

Implicit complexity has been very successful in capturing lots of different complexity classes! But the programming languages involved are often ad-hoc…

> *Several systems* […] *have been produced; my favourite being* **LLL**, *light linear logic, which* […] *can harbour all polytime functions. Unfortunately these systems are good for nothing, they all come from bondage: artificial restrictions on the rules which achieve certain effects, but are not justified by use, not even by some natural "semantic" considerations.* — Girard, *From Foundations to Ludics*

Yet *we didn't ask* for regular languages to appear in the simply typed λ-calculus!

"Implicit automata" challenge: find *natural* characterizations for other automata-theoretic classes of languages/functions using typed λ-calculi

Grandeur et misère de la complexité implicite

Implicit complexity has been very successful in capturing lots of different complexity classes! But the programming languages involved are often ad-hoc...

> *Several systems* [...] *have been produced; my favourite being* **LLL**, *light linear logic, which* [...] *can harbour all polytime functions. Unfortunately these systems are good for nothing, they all come from bondage: artificial restrictions on the rules which achieve certain effects, but are not justified by use, not even by some natural "semantic" considerations.* — Girard, *From Foundations to Ludics*

Yet *we didn't ask* for regular languages to appear in the simply typed λ-calculus!

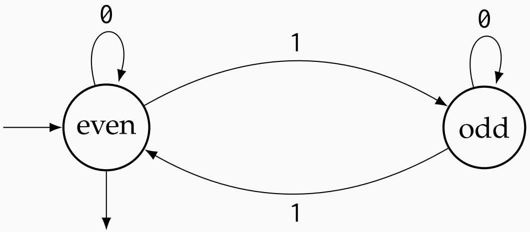
"Implicit automata" challenge: find *natural* characterizations for other automata-theoretic classes of languages/functions using typed λ-calculi

Next: we review regular languages and *star-free languages*, our new target

Regular languages

Many classical equivalent definitions (+ STλC with Church encodings!):

- *regular expressions*: `0*(10*10*)*` = "only `0`s and `1`s & even number of `1`s"
- *finite automata* (DFA/NFA): e.g. drawing below



Regular languages

Many classical equivalent definitions (+ STλC with Church encodings!):

- *regular expressions*: `0*(10*10*)*` = "only `0`s and `1`s & even number of `1`s"
- *finite automata* (DFA/NFA)
- *algebraic* definition below (very close to DFA), e.g. $M = \mathbb{Z}/(2)$

Theorem (classical)

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

Σ: finite alphabet, Σ^*: words over Σ
monoid structure: for $v, w \in \Sigma^*$, $v \cdot w$ = concatenation
morphism: for $w \in \Sigma^*$ with n letters, $\varphi(w) = \varphi(w[0]) \dots \varphi(w[n])$

Star-free languages and aperiodicity

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

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

e.g. $(ab)^* = (b\varnothing^c \cup \varnothing^c a \cup \varnothing^c aa\varnothing^c \cup \varnothing^c bb\varnothing^c)^c$

Star-free languages and aperiodicity

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

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

e.g. $(ab)^* = (b\varnothing^c \cup \varnothing^c a \cup \varnothing^c aa\varnothing^c \cup \varnothing^c bb\varnothing^c)^c$ but $(aa)^*$ is not star-free...

Star-free languages and aperiodicity

<u>**Star-free languages**</u>: regular expressions with *complementation* but *without star*

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

e.g. $(ab)^* = (b\varnothing^c \cup \varnothing^c a \cup \varnothing^c aa\varnothing^c \cup \varnothing^c bb\varnothing^c)^c$ but $(aa)^*$ is not star-free...

Theorem (classical)

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

Theorem (Schützenberger 1965)

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

Star-free languages and aperiodicity

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

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

e.g. $(ab)^* = (b\varnothing^c \cup \varnothing^c a \cup \varnothing^c aa \varnothing^c \cup \varnothing^c bb \varnothing^c)^c$ but $(aa)^*$ is not star-free...

Definition

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

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

Theorem (Schützenberger 1965)

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

From aperiodicity to non-commutativity

How to enforce aperiodicity in a λ-calculus? Consider monoids of terms $t : A \to A$

Embedding of non-aperiodic $\mathbb{Z}/(2)$ via $\texttt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\texttt{not} \circ \texttt{not} =_\beta \texttt{id}$)

From aperiodicity to non-commutativity

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

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

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

From aperiodicity to non-commutativity

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

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

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

Idea: non-commutative type system, i.e. make the order of arguments matter

"a function $\lambda b.\ \lambda x.\ \lambda y.\ (\dots)$ should first use b, then x, then y"

From aperiodicity to non-commutativity

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

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

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

Idea: non-commutative type system, i.e. make the order of arguments matter

"a function $\lambda b.\ \lambda x.\ \lambda y.\ (\ldots)$ should first use b, then x, then y"

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

From aperiodicity to non-commutativity

How to enforce aperiodicity in a λ-calculus? Consider monoids of terms $t : A \to A$

Embedding of non-aperiodic $\mathbb{Z}/(2)$ via $\texttt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\texttt{not} \circ \texttt{not} =_\beta \texttt{id}$)

$$\texttt{true} = \lambda x.\, \lambda y.\, x \qquad \texttt{false} = \lambda x.\, \lambda y.\, y \qquad \texttt{not} = \lambda b.\, \lambda x.\, \lambda y.\, b\, y\, x$$

morally, $\texttt{if } b \texttt{ then } x \texttt{ else } y \rightsquigarrow \texttt{if not}(b) \texttt{ then } y \texttt{ else } x$

the `not` function *exchanges* two of its arguments

Idea: non-commutative type system, i.e. make the order of arguments matter

"a function $\lambda b.\, \lambda x.\, \lambda y.\, (\dots)$ should first use b, then x, then y"

Technical issue: $\lambda f.\, \lambda x.\, \lambda y.\, (\lambda z.\, f\, z\, z)\, (x\, y) \longrightarrow_\beta \lambda f.\, \lambda x.\, \lambda y.\, f\, (x\, y)\, (x\, y)$

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

Non-commutative types and linear logic

Idea: *non-commutative type system*, i.e. make the order of arguments matter

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\, z\, z)\ (x\ y) \longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$

the problem comes from the *two copies* of $(x\ y)$, caused by two occurrences of z

Non-commutative types and linear logic

Idea: *non-commutative type system*, i.e. make the order of arguments matter

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\, z\, z)\ (x\ y) \longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$

the problem comes from the *two copies* of $(x\ y)$, caused by two occurrences of z

Fix: prohibit duplication $\longrightarrow$ non-commutative <u>affine</u> λ-calculus

"a function should use its argument *at most once*"

Non-commutative types and linear logic

Idea: *non-commutative type system*, i.e. make the order of arguments matter

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\ z\ z)\ (x\ y) \longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$

the problem comes from the *two copies* of $(x\ y)$, caused by two occurrences of z

Fix: prohibit duplication $\longrightarrow$ non-commutative affine λ-calculus

"a function should use its argument *at most once*"

If "*exactly once*", non-commutative *linear* λ-calculus; an old idea:

- first introduced by Lambek (1958), applied to linguistics
- revival in late 1980s with the birth of *linear logic* (Girard)
- recently: correspondence with planar combinatorial maps (N. Zeilberger)

Non-commutative types and linear logic

Idea: *non-commutative type system*, i.e. make the order of arguments matter

Technical issue: $\lambda f.\ \lambda x.\ \lambda y.\ (\lambda z.\ f\, z\, z)\ (x\ y) \longrightarrow_\beta \lambda f.\ \lambda x.\ \lambda y.\ f\ (x\ y)\ (x\ y)$

the problem comes from the *two copies* of $(x\ y)$, caused by two occurrences of z

Fix: prohibit duplication $\longrightarrow$ non-commutative <u>affine</u> λ-calculus

"a function should use its argument *at most once*"

If "*exactly once*", non-commutative *linear* λ-calculus; an old idea:

- first introduced by Lambek (1958), applied to linguistics
- revival in late 1980s with the birth of *linear logic* (Girard)
- recently: correspondence with planar combinatorial maps (N. Zeilberger)

$\longrightarrow$ not contrived to get a connection with automata!

Finally, our theorem

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

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

A function $\lambda^{\circ}x.\ \lambda^{\to}y.\ \lambda^{\circ}z.\ (\dots)$ can use each of x and z at most once

cannot use x after z no restrictions on y

Finally, our theorem

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

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

A function $\lambda^{\circ}x.\ \lambda^{\to}y.\ \lambda^{\circ}z.\ (\dots)$ can use each of x and z at most once

cannot use x after z no restrictions on y

Church encoding with affine types

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

Finally, our theorem

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

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

A function $\lambda^{\circ}x.\ \lambda^{\rightarrow}y.\ \lambda^{\circ}z.\ (\dots)$ can use each of x and z at most once

cannot use x after z no restrictions on y

Church encoding with affine types

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

Theorem (N. & Pradic, ICALP 2020)

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

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

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

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

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

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

$$\frac{\Gamma \mid \Delta \cdot (x : A) \vdash t : B}{\Gamma \mid \Delta \vdash \lambda^{\circ} x.\, t : A \multimap B} \qquad \frac{\Gamma \mid \Delta \vdash t : A}{\Gamma \mid \Delta' \vdash t : A} \text{ when } \Delta \text{ is a } \textbf{subsequence} \text{ of } \Delta'$$

without weakening (last rule) $\approx$ Polakow & Pfenning's Intuitionistic Non-Commutative Linear Logic

Remarks on the proof

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

Lemma (in our non-commutative λ-calculus)

For any purely affine A, the monoid $\{t \mid t : A \multimap A\}/=_{\beta\eta}$ is finite and aperiodic.

Finite due to affineness, aperiodic due to non-commutativity.

Remarks on the proof

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

Lemma (in our non-commutative λ-calculus)

For any purely affine A, the monoid $\{t \mid t : A \multimap A\}/{=_{\beta\eta}}$ is finite and aperiodic.

Finite due to affineness, aperiodic due to non-commutativity.

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

Remarks on the proof

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

Lemma (in our non-commutative λ-calculus)

For any purely affine A, the monoid $\{t \mid t : A \multimap A\}/=_{\beta\eta}$ is finite and aperiodic.

Finite due to affineness, aperiodic due to non-commutativity.

The converse is harder (unusual for implicit complexity!): how do we exploit the aperiodicity assumption? Using the powerful toolbox of finite semigroup theory

Theorem (Krohn & Rhodes 1965 (special case))

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

Remarks on the proof

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

Lemma (in our non-commutative λ-calculus)

For any purely affine A, the monoid $\{t \mid t : A \multimap A\}/{=_{\beta\eta}}$ is finite and aperiodic.

Finite due to affineness, aperiodic due to non-commutativity.

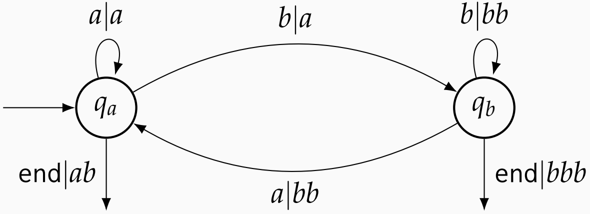
The converse is harder (unusual for implicit complexity!): how do we exploit the aperiodicity assumption? Using the powerful toolbox of finite semigroup theory

Theorem (Krohn & Rhodes 1965 (special case))

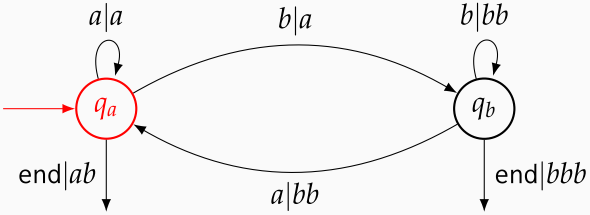
Any finite and aperiodic monoid can be "decomposed" as a wreath product *of "building blocks" which are certain monoids with 3 elements.*

To avoid the scary algebra: a detour through *transducers*, i.e. automata with output.

Sequential transducers: an example

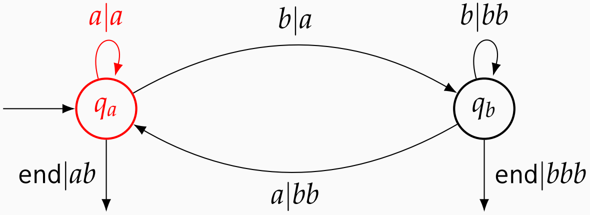


Sequential transducers: an example



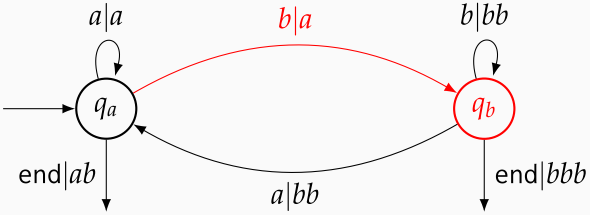
$$abba \mapsto$$

Sequential transducers: an example



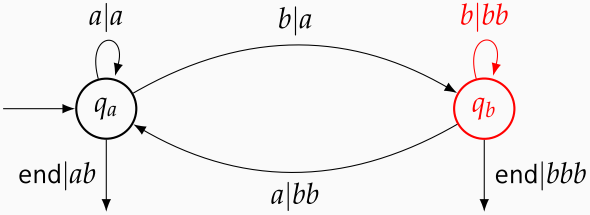
$abba \mapsto a$

Sequential transducers: an example



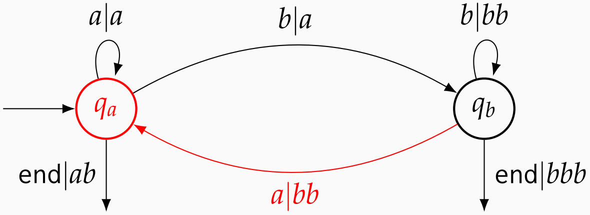
$$abba \mapsto aa$$

Sequential transducers: an example



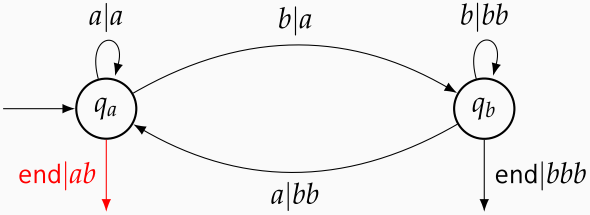
$$abba \mapsto aabb$$

Sequential transducers: an example



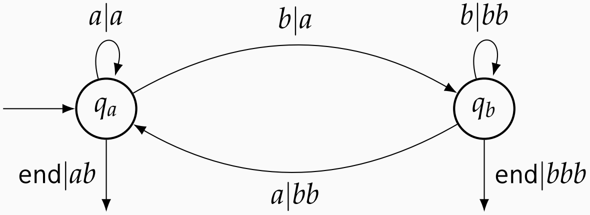
$$abba \mapsto aabbbb$$

Sequential transducers: an example



$$abba \mapsto aabbbbab$$

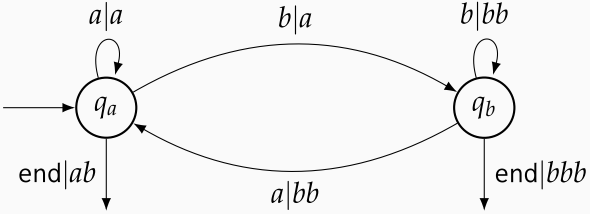
Sequential transducers: an example



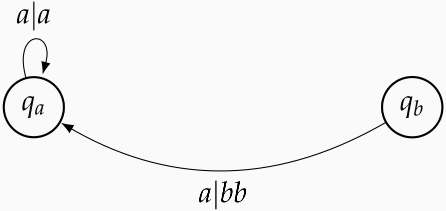
$$abba \mapsto aabbbbab$$

Note: there is an implicit characterization of sequential transductions using *cyclic proofs* [DeYoung & Pfenning, APLAS'16]; Anupam and Gianluca's talks later today will show other results on ICC with cyclic proofs

Aperiodic sequential transducers

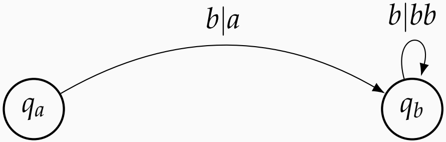


Aperiodic sequential transducers



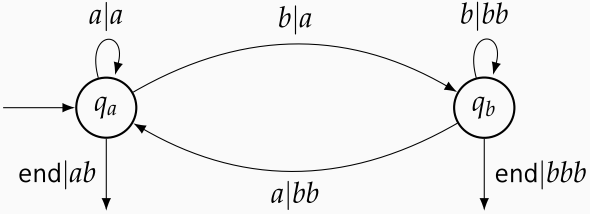
Transition for a: $\_ \mapsto q_a$

Aperiodic sequential transducers



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

Aperiodic sequential transducers



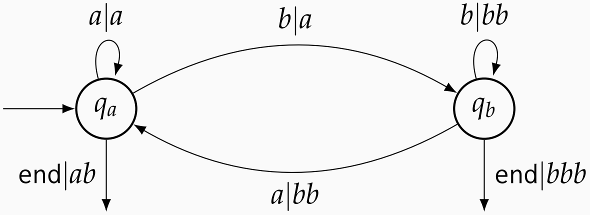
Transition for a: $\_ \mapsto q_a$ Transition for b: $\_ \mapsto q_b$

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

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

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

Aperiodic sequential transducers



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

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

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

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

Lemma

$L \subseteq \Sigma^*$ *star-free* $\iff$ $L = f^{-1}(\varepsilon)$ *for some aperiodic sequential* $f : \Sigma^* \to \Gamma^*$

The Krohn–Rhodes decomposition, again

Reformulation of the Krohn–Rhodes decomposition

Aperiodic sequential functions are generated from aper. seq. transducers *with 2 states* (as in prev. slide) by usual function composition.

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

The Krohn–Rhodes decomposition, again

Reformulation of the Krohn–Rhodes decomposition

Aperiodic sequential functions are generated from aper. seq. transducers *with 2 states* (as in prev. slide) by usual function composition.

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

Theorem

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

Corollary

It can define all star-free languages with terms of type $\mathsf{Str}_\Sigma[A] \multimap \mathsf{Bool}$.

String-to-string functions

Theorem

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

Obtained as byproduct of our proof. What about the converse?

String-to-string functions

Theorem

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

Obtained as byproduct of our proof. What about the converse?

False: we can code non-sequential functions, e.g. `reverse` $: \mathsf{Str}_\Sigma[o \multimap o] \multimap \mathsf{Str}_\Sigma$.

- Exact characterization of $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (A purely affine)?
- What happens in a commutative affine λ-calculus?

String-to-string functions

Theorem

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

Obtained as byproduct of our proof. What about the converse?
False: we can code non-sequential functions, e.g. `reverse` $: \mathsf{Str}_\Sigma[o \multimap o] \multimap \mathsf{Str}_\Sigma$.

- Exact characterization of $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (A purely affine)?
- What happens in a commutative affine λ-calculus?

Similar to questions at the beginning about simply typed λ-calculus,
but affineness makes things easier.

Characterizing regular functions

Theorem (commutative case)

$f : \Gamma^* \to \Sigma^*$ can be expressed by an affine λ-term $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (A purely affine) $\iff$ f is a regular function.

Regular functions admit many equivalent definitions; among others:

- monadic second-order logic (reg. fn. also called "MSO transductions")
- basic functions + combinators (several variants)
- copyless streaming string transducers
- *two-way* finite state transducers

Non-commutative case: *aperiodic regular functions* ("first-order transductions")

Characterizing regular functions

Theorem (commutative case)

$f : \Gamma^* \to \Sigma^*$ can be expressed by an affine λ-term $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (A purely affine)
$\iff$ *f is a* regular function.

Regular functions admit many equivalent definitions; among others:

- monadic second-order logic (reg. fn. also called "MSO transductions")
- basic functions + combinators (several variants)
- **copyless** streaming string transducers $\simeq$ affine types!
- *two-way* finite state transducers

Non-commutative case: *aperiodic regular functions* ("first-order transductions")

Characterizing regular functions

Theorem (commutative case)

$f : \Gamma^* \to \Sigma^*$ *can be expressed by an affine* λ*-term* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *(*A *purely affine)*
$\iff$ f *is a* regular function.

Regular functions admit many equivalent definitions; among others:

- monadic second-order logic (reg. fn. also called "MSO transductions")
- basic functions + combinators (several variants)
- **copyless** streaming string transducers $\simeq$ affine types!
- *two-way* finite state transducers, closely related to the *geometry of interaction* semantics of linear logic [Hines 2003]

Non-commutative case: *aperiodic regular functions* ("first-order transductions")

Decomposition of regular functions

One possible proof regular function $\implies$ affine λ-definable uses:

Theorem (Bojańczyk et al. (see e.g. [Bojańczyk & Stefański 2020]))

Any regular function can be obtained as a composition of:

- *sequential functions (that can themselves be decomposed by Krohn–Rhodes);*
- $\mathtt{mapReverse}_\Sigma, \mathtt{mapDuplicate}_\Sigma : (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ *for* $\# \notin \Sigma$

For $w_1, \ldots, w_n \in \Sigma^*$,

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

Decomposition of regular functions

One possible proof regular function $\implies$ affine λ-definable uses:

Theorem (Bojańczyk et al. (see e.g. [Bojańczyk & Stefański 2020]))

Any regular function can be obtained as a composition of:

- *sequential functions* (*that can themselves be decomposed by Krohn–Rhodes*);
- $\texttt{mapReverse}_\Sigma, \texttt{mapDuplicate}_\Sigma : (\Sigma \cup \{\#\})^* \to (\Sigma \cup \{\#\})^*$ *for* $\# \notin \Sigma$

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

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

Such factorization theorems (the above and Krohn–Rhodes) are a form of compositional (algebraic?) "structure" arising in the study of "power"!
This often happens in automata theory...

Semantic evaluation example: STλC-definable languages are regular

Next: a technique to bound the expressive power of λ-terms.

Example: for any type A & any simply typed λ-term $t : \mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A] \to \mathsf{Bool}$, the language $\{w \in \{0,1\}^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is *regular* [Hillebrand & Kanellakis 1996].

Semantic evaluation example: STλC-definable languages are regular

Next: a technique to bound the expressive power of λ-terms.

Example: for any type A & any simply typed λ-term $t : \mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A] \to \mathsf{Bool}$, the language $\{w \in \{0,1\}^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is *regular* [Hillebrand & Kanellakis 1996].

Proof sketch: by semantic evaluation.

Use the standard semantics in *finite sets*: $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$.

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

Semantic evaluation example: STλC-definable languages are regular

Next: a technique to bound the expressive power of λ-terms.

Example: for any type A & any simply typed λ-term $t : \mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A] \to \mathsf{Bool}$, the language $\{w \in \{0,1\}^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is *regular* [Hillebrand & Kanellakis 1996].

Proof sketch: by <u>semantic evaluation</u>.

Use the standard semantics in *finite sets*: $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$.

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

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

Semantic evaluation example: STλC-definable languages are regular

Next: a technique to bound the expressive power of λ-terms.

Example: for any type A & any simply typed λ-term $t : \mathsf{Str}_{\{\mathtt{0},\mathtt{1}\}}[A] \to \mathsf{Bool}$, the language $\{w \in \{0,1\}^* \mid t\,\overline{w} =_\beta \mathtt{true}\}$ is *regular* [Hillebrand & Kanellakis 1996].

Proof sketch: by semantic evaluation.

Use the standard semantics in *finite sets*: $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$.

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

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

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

Categorical automata theory meets semantic evaluation

Semantic evaluation strategy for affine λ-definable $\implies$ regular function:

- Consider a *category* $\mathcal{C}$ of "transducer behaviors", such that *automata over* $\mathcal{C}$ (in the sense of [Colcombet & Petrişan 2017]) compute regular functions
  - $\mathcal{C} = \mathsf{Int}(\mathsf{PFinSet})$ (geometry of interaction): two-way transducers
  - $\mathcal{C}$ = "Dialectica-like" category of affine register assignments: variant of copyless streaming string transducers
  - this reflects compositional *structure* that is actually used in "mainstream" automata theory, under the guise of "monoids of behaviors"!

Categorical automata theory meets semantic evaluation

Semantic evaluation strategy for affine λ-definable $\implies$ regular function:

- Consider a *category* $\mathcal{C}$ of "transducer behaviors", such that *automata over* $\mathcal{C}$ (in the sense of [Colcombet & Petrişan 2017]) compute regular functions
  - $\mathcal{C} = \mathsf{Int}(\mathsf{PFinSet})$ (geometry of interaction): two-way transducers
  - $\mathcal{C}$ = "Dialectica-like" category of affine register assignments: variant of copyless streaming string transducers
  - this reflects compositional *structure* that is actually used in "mainstream" automata theory, under the guise of "monoids of behaviors"!
- Exhibit a *monoidal closed* structure on $\mathcal{C}$
- Conclude by interpreting purely affine λ-terms in $\mathcal{C}$

Categorical automata theory meets semantic evaluation

Semantic evaluation strategy for affine λ-definable $\implies$ regular function:

- Consider a *category* $\mathcal{C}$ of "transducer behaviors", such that *automata over* $\mathcal{C}$ (in the sense of [Colcombet & Petrişan 2017]) compute regular functions
  - $\mathcal{C} = \mathsf{Int}(\mathsf{PFinSet})$ (geometry of interaction): two-way transducers
  - $\mathcal{C}$ = "Dialectica-like" category of affine register assignments: variant of copyless streaming string transducers
  - this reflects compositional *structure* that is actually used in "mainstream" automata theory, under the guise of "monoids of behaviors"!
- Exhibit a *monoidal closed* structure on $\mathcal{C}$
- Conclude by interpreting purely affine λ-terms in $\mathcal{C}$

Side benefits: some composition and determinization theorems for transducers "secretly rely on" monoidal closed categories, leading to generalizations

Perspectives: new transducer models

- *Comparison-free polyregular functions* [N., Noûs, Pradic ICALP'21]: discovered by playing around with $\mathsf{Str}[A] \to \mathsf{Str}$ instead of $\mathsf{Str}[A] \multimap \mathsf{Str}$ natural from an automata-theoretic POV, part of a recent line of investigations into polynomial growth transductions (Bojańczyk, Douéneau, Kiefer, Lhote, ...)

Perspectives: new transducer models

- *Comparison-free polyregular functions* [N., Noûs, Pradic ICALP'21]:
  discovered by playing around with $\mathsf{Str}[A] \to \mathsf{Str}$ instead of $\mathsf{Str}[A] \multimap \mathsf{Str}$
  natural from an automata-theoretic POV, part of a recent line of investigations into polynomial growth transductions (Bojańczyk, Douéneau, Kiefer, Lhote, ...)
- Two-way automata/transducers with *planar behaviors*:
  new machine model for star-free languages / aperiodic regular functions
  inspiration topological planarity $\leftrightarrow$ non-commutative types

Perspectives: new transducer models

- *Comparison-free polyregular functions* [N., Noûs, Pradic ICALP'21]:
  discovered by playing around with $\mathsf{Str}[A] \to \mathsf{Str}$ instead of $\mathsf{Str}[A] \multimap \mathsf{Str}$
  natural from an automata-theoretic POV, part of a recent line of investigations into polynomial growth transductions (Bojańczyk, Douéneau, Kiefer, Lhote, ...)
- Two-way automata/transducers with *planar behaviors*:
  new machine model for star-free languages / aperiodic regular functions
  inspiration topological planarity $\leftrightarrow$ non-commutative types
- "Collapsible pushdown transducers": answer to old question on STλC?
  Lessons from 1980s literature on tree transducers
  + 2010s work on higher-order model checking

Perspectives: new transducer models

- *Comparison-free polyregular functions* [N., Noûs, Pradic ICALP'21]:
  discovered by playing around with $\mathsf{Str}[A] \to \mathsf{Str}$ instead of $\mathsf{Str}[A] \multimap \mathsf{Str}$
  natural from an automata-theoretic POV, part of a recent line of investigations into polynomial growth transductions (Bojańczyk, Douéneau, Kiefer, Lhote, ...)
- Two-way automata/transducers with *planar behaviors*:
  new machine model for star-free languages / aperiodic regular functions
  inspiration topological planarity $\leftrightarrow$ non-commutative types
- "Collapsible pushdown transducers": answer to old question on STλC?
  Lessons from 1980s literature on tree transducers
  + 2010s work on higher-order model checking

Also: tree automata vs multiplicative/additive distinction in linear logic

By way of conclusion: back to Structure meets Power

There is a remarkable divide in the field of logic in Computer Science, between two distinct strands: one focusing on semantics and compositionality ("Structure"), the other on expressiveness and complexity ("Power"). It is remarkable because these two fundamental aspects of our field are studied using almost disjoint technical languages and methods, by almost disjoint research communities.

By way of conclusion: back to Structure meets Power

There is a remarkable divide in the field of logic in Computer Science, between two distinct strands: one focusing on semantics and compositionality ("Structure"), the other on expressiveness and complexity ("Power"). It is remarkable because these two fundamental aspects of our field are studied using almost disjoint technical languages and methods, by almost disjoint research communities. **We believe that bridging this divide is a major issue in Computer Science, and may hold the key to fundamental advances in the field.**

By way of conclusion: back to Structure meets Power

There is a remarkable divide in the field of logic in Computer Science, between two distinct strands: one focusing on semantics and compositionality ("Structure"), the other on expressiveness and complexity ("Power"). It is remarkable because these two fundamental aspects of our field are studied using almost disjoint technical languages and methods, by almost disjoint research communities. **We believe that bridging this divide is a major issue in Computer Science, and may hold the key to fundamental advances in the field.**

Thanks for your attention! Questions?