Implicit automata in typed λ-calculi

Lê Thành Dũng (Tito) Nguyễn (École normale supérieure de Lyon)
joint work with **Cécilia Pradic** (Swansea University)

Oxford Advanced Seminar on Informatic Structures (OASIS), 9th June 2023

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.

— S. Abramsky (from the blurb of the *Structure meets Power* workshops)

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.

— S. Abramsky (from the blurb of the *Structure meets Power* workshops)

- OASIS seminar: mostly about "structure"
- This talk: connections with automata, from the "power" side
    - Are they really though? I'll come back to that during the talk

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! 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! 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! 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*: $\mathsf{Nat} = (o \to o) \to o \to o$

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

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

Simply typed functions on Church numerals

Church numerals: $\mathsf{Nat} = (o \to o) \to o \to o$

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

Simply typed functions on Church numerals

Church numerals: $\mathsf{Nat} = (o \to o) \to o \to o$ $\qquad$ $\mathsf{Nat}[A/o] = (A \to A) \to A \to A$

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

Let's add a bit of (meta-level) polymorphism: $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o]$ for $n \in \mathbb{N}$

More difficult question (what is the right perspective on it?)

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

Church numerals: $\mathsf{Nat} = (o \to o) \to o \to o$ $\qquad\qquad$ $\mathsf{Nat}[A/o] = (A \to A) \to A \to A$

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

Let's add a bit of (meta-level) polymorphism: $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o]$ for $n \in \mathbb{N}$

More difficult question (what is the right perspective on it?)

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?

- Looks weird: you can express towers of exponentials, but not subtraction or equality (Statman 198X) – is it a good question?
- Not so important: this is about "power" while our focus is on "structure"

Simply typed functions on Church numerals

Church numerals: $\mathsf{Nat} = (o \to o) \to o \to o$ $\quad\quad \mathsf{Nat}[A/o] = (A \to A) \to A \to A$

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

Let's add a bit of (meta-level) polymorphism: $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o]$ for $n \in \mathbb{N}$

More difficult question (what is the right perspective on it?)

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?

- Looks weird: you can express towers of exponentials, but not subtraction or equality (Statman 198X) – is it a good question?
- 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; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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; $\mathsf{Nat} = \mathsf{Str}_{\{1\}}$):

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

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

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

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

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

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

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

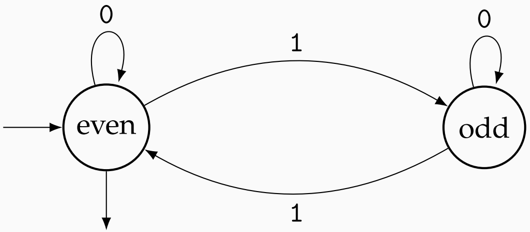
Theorem (Hillebrand & Kanellakis 1996)

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

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 $\varphi\colon \Sigma^* \to M$ *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, $\varphi(w) = \varphi(w[0]) \ldots \varphi(w[n])$

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 $\mathcal{L}(t) = \{w \in \Sigma^* \mid t\,\overline{w} \to_\beta^* \mathtt{true}\}$ is regular.*

Part 1 of proof.

Fix type A. Any *denotational semantics* $[\![-]\!]$ quotients words:

$$w \in \Sigma^* \rightsquigarrow \overline{w} : \mathsf{Str}[A] \rightsquigarrow [\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]} \in [\![\mathsf{Str}_\Sigma[A]]\!]$$

$[\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]}$ determines behavior of w w.r.t. all $\mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ terms:

$$w \in \mathcal{L}(t) \iff t\,\overline{w} \to_\beta^* \mathtt{true} \underbrace{\iff}_{\text{assuming } [\![\mathtt{true}]\!] \neq [\![\mathtt{false}]\!]} [\![t\,\overline{w}]\!] = [\![t]\!]([\![\overline{w}]\!]) = [\![\mathtt{true}]\!]$$

Goal: to decide $\mathcal{L}(t)$, compute $w \mapsto [\![\overline{w}]\!]$ in some denotational model.

Proof of STλC-definable $\implies$ regular (the converse is easy)

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 $\mathcal{L}(t) = \{w \in \Sigma^* \mid t\,\overline{w} \to_\beta^* \mathtt{true}\}$ is regular.

Part 2 of proof.

We use $[\![-]\!] : \mathrm{ST}\lambda\mathrm{C} \to \mathsf{FinSet}$ to build a DFA with states $Q = [\![\mathsf{Str}_\Sigma[A]]\!]$, acceptation as $[\![t]\!](-) = [\![\mathtt{true}]\!]$.

$$\longrightarrow [\![\overline{\varepsilon}]\!] \xrightarrow{\mathtt{a}} [\![\overline{\mathtt{a}}]\!] \xrightarrow{\mathtt{b}} [\![\overline{\mathtt{ab}}]\!] \xrightarrow{\mathtt{b}} [\![\overline{\mathtt{abb}}]\!] \longrightarrow \cdots$$

$$w \in \mathcal{L}(t) \iff [\![t]\!]\left([\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]}\right) = [\![\mathtt{true}]\!] \iff w \text{ accepted}$$

$\longrightarrow$ *semantic evaluation* argument (variant: morphism to monoid $[\![\mathsf{Str}_\Sigma[A]]\!]$) $\square$

Proof of STλC-definable $\implies$ regular (the converse is easy)

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 $\mathcal{L}(t) = \{w \in \Sigma^* \mid t\,\overline{w} \to_\beta^* \mathtt{true}\}$ is regular.

Part 2 of proof.

We use $[\![-]\!] : \mathrm{ST}\lambda\mathrm{C} \to \mathsf{FinSet}$ to build a DFA with states $Q = [\![\mathsf{Str}_\Sigma[A]]\!]$, acceptation as $[\![t]\!](-) = [\![\mathtt{true}]\!]$. $\quad$ ($|Q| < \infty$, e.g. $2^{2^{134}}$ when $A = \mathsf{Bool}$ & $|\,[\![o]\!]\,| = 2 = |\Sigma|$)

$$\longrightarrow [\![\overline{\varepsilon}]\!] \xrightarrow{\mathtt{a}} [\![\overline{\mathtt{a}}]\!] \xrightarrow{\mathtt{b}} [\![\overline{\mathtt{ab}}]\!] \xrightarrow{\mathtt{b}} [\![\overline{\mathtt{abb}}]\!] \longrightarrow \cdots$$

$$w \in \mathcal{L}(t) \iff [\![t]\!]\left([\![\overline{w}]\!]_{\mathsf{Str}_\Sigma[A]}\right) = [\![\mathtt{true}]\!] \iff w \text{ accepted}$$

$\longrightarrow$ *semantic evaluation* argument (variant: morphism to monoid $[\![\mathsf{Str}_\Sigma[A]]\!]$) $\square$

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*

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*
- *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*
- *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

Grandeur et misère de la complexité implicite

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.* — J.-Y. Girard, *From Foundations to Ludics*

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.* — J.-Y. 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.* — J.-Y. 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

Our new target: the class of *star-free languages* (we'll come back to $\mathbb{N} \to \mathbb{N}$ later)

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*

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

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 $\varphi\colon \Sigma^* \to M$ *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 $\varphi\colon \Sigma^* \to M$ 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 $\varphi \colon \Sigma^* \to M$ 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 `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 $\texttt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\texttt{not} \circ \texttt{not} =_\beta \texttt{id}$)

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

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

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

"a function $\lambda b.\, \lambda x.\, \lambda y.\, (\ldots)$ 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 $\texttt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\texttt{not} \circ \texttt{not} =_\beta \texttt{id}$)

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

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

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

"a function $\lambda b.\, \lambda x.\, \lambda y.\, (\ldots)$ 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 $\mathtt{not} : \mathsf{Bool} \to \mathsf{Bool}$ ($\mathtt{not} \circ \mathtt{not} =_\beta \mathtt{id}$)

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

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 $\lambda b.\, \lambda x.\, \lambda y.\, (\dots)$ 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 affine 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 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)

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

Finally, our theorem: a computational consequence of non-commutative typing

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

cannot use x after z no restrictions on y

Finally, our theorem: a computational consequence of non-commutative typing

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

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

Finally, our theorem: a computational consequence of non-commutative typing

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

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

Theorem (N. & Pradic 2020 + linear instead of affine variant in my PhD)

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 '$\to$'.*

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

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

A non-commutative affine type system

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

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

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

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

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

Remarks on the proof

To prove "non-commutatively 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 "non-commutatively 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 "non-commutatively 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 (special case of [Krohn & Rhodes 1965])

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 "non-commutatively 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 (special case of [Krohn & Rhodes 1965])

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.

Structure in the service of Power: applying a factorization theorem

The Krohn–Rhodes decomposition rephrased

The class of *aperiodic sequential functions* is generated from very simple string-to-string transducers (with 2 states) by usual function composition.

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

Structure in the service of Power: applying a factorization theorem

The Krohn–Rhodes decomposition rephrased

The class of *aperiodic sequential functions* is generated from very simple string-to-string transducers (with 2 states) by usual function composition.

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

Theorem

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

Proof: it's enough to find λ-terms for the "building block" transducers

(not-so-trivial programming exercise!)

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 at least all aperiodic sequential functions with terms $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (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 at least all aperiodic sequential functions with terms $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (A purely affine).

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

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

(sequential functions are "left-to-right")

- Exact characterization of $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (A purely affine)?
- What happens in a commutative affine λ-calculus?
  At least all (not necessarily aperiodic) sequential functions; actually more

String-to-string functions

Theorem

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

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

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

(sequential functions are "left-to-right")

- Exact characterization of $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ (A purely affine)?
- What happens in a commutative affine λ-calculus?
  At least all (not necessarily aperiodic) sequential functions; actually more

Similar to questions at the beginning about simply typed λ-calculus (in the case $\mathbb{N} \to \mathbb{N}$) but affineness makes things easier.

Characterizing regular functions

Theorem

$f : \Gamma^* \to \Sigma^*$ 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 (*commutative case*) / aperiodic reg. fn. (*non-comm. case*)

e.g. $\textsf{map-copy-reverse}(aab\#abc\#\dots) = aab\#baa\#abc\#cba\#\dots$
Regular functions admit many equivalent definitions; among others:

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

Characterizing regular functions

Theorem

$f : \Gamma^* \to \Sigma^*$ 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 (*commutative case*) / aperiodic reg. fn. (*non-comm. case*)

e.g. $\mathsf{map\text{-}copy\text{-}reverse}(aab\#abc\#\ldots) = aab\#baa\#abc\#cba\#\ldots$
Regular functions admit many equivalent definitions; among others:

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

$\downarrow$

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

$\downarrow$

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

$$\downarrow$$

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

$$\downarrow$$

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

$$\downarrow$$

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

$$\downarrow$$

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

$$\downarrow$$

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010] a.k.a. register transducers

Deterministic finite state automaton + string-valued *registers*. Example:

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

a	c	a	b	$\#$	b	c	$\#$	c	a

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

Regular functions = computed by copyless SSTs

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

each register appears at most once on the right of a := in a transition

Proof technique for affinely λ-definable $\implies$ regular function

As in [Hillebrand & Kanellakis 1996] for STλC, we use semantic evaluation

$\mathcal{C}$ = "Dialectica-like" variant of the *category* of copyless register updates

- $\mathcal{C}$ is (affine) *monoidal closed*: provides a semantics for purely affine λ-terms
- and $\underbrace{\textit{automata over } \mathcal{C}}_{\text{in the sense of [Colcombet \& Petrişan 2017]}}$ compute exactly the regular functions

Proof technique for affinely λ-definable $\implies$ regular function

As in [Hillebrand & Kanellakis 1996] for STλC, we use <u>semantic evaluation</u>

$\mathcal{C}$ = "Dialectica-like" variant of the *category* of copyless register updates

- $\mathcal{C}$ is (affine) *monoidal closed*: provides a semantics for purely affine λ-terms
- and $\underbrace{\textit{automata over } \mathcal{C}}_{\text{in the sense of [Colcombet \& Petrişan 2017]}}$ compute exactly the regular functions

This reflects compositional *structure* that is actually used in "mainstream" automata theory, under the guise of "monoids of behaviors"!

Proof technique for affinely λ-definable $\implies$ regular function

As in [Hillebrand & Kanellakis 1996] for STλC, we use semantic evaluation

$\mathcal{C}$ = "Dialectica-like" variant of the *category* of copyless register updates

- $\mathcal{C}$ is (affine) *monoidal closed*: provides a semantics for purely affine λ-terms
- and $\underbrace{\textit{automata over } \mathcal{C}}_{\text{in the sense of [Colcombet \& Petrişan 2017]}}$ compute exactly the regular functions

This reflects compositional *structure* that is actually used in "mainstream" automata theory, under the guise of "monoids of behaviors"!

Looking back at this a few years later...

Perhaps the main use of monoidal closure is to form the *internal monoids* $X \multimap X$

$\leadsto$ inspired a very concise monoid-based categorical definition of regular functions [Bojańczyk & N., ICALP 2023]

A big technical digression

Automata over the category $\mathsf{Int}(\mathsf{PFinSet})$ = two-way transducers [Hines 2003]
$\rightsquigarrow$ related to the "geometry of interaction" semantics of linear logic; drawbacks:

- not affine
- no additive connectives $\&/\oplus$

A big technical digression

Automata over the category $\mathsf{Int}(\mathsf{PFinSet})$ = two-way transducers [Hines 2003]
$\rightsquigarrow$ related to the "geometry of interaction" semantics of linear logic; drawbacks:

- not affine: superficial issue, just use the Interaction Abstract Machine instead (ongoing work with Gabriele Vanoni)
- no additive connectives $\&/\oplus$

A big technical digression

Automata over the category $\mathsf{Int}(\mathsf{PFinSet})$ = two-way transducers [Hines 2003]
$\rightsquigarrow$ related to the "geometry of interaction" semantics of linear logic; drawbacks:

- not affine: superficial issue, just use the Interaction Abstract Machine instead
  (ongoing work with Gabriele Vanoni)
- no additive connectives $\&/\oplus$: a feature rather than a bug, allows proving
  $\mathsf{Tree}_\Sigma[A] \multimap \mathsf{Bool}$ w/o $\&/\oplus \subseteq$ tree-walking automata $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ reg. tree languages

A big technical digression

Automata over the category $\mathsf{Int}(\mathsf{PFinSet})$ = two-way transducers [Hines 2003]
$\rightsquigarrow$ related to the "geometry of interaction" semantics of linear logic; drawbacks:

- not affine: superficial issue, just use the Interaction Abstract Machine instead
  (ongoing work with Gabriele Vanoni)
- no additive connectives $\&/\oplus$: a feature rather than a bug, allows proving
  $\mathsf{Tree}_\Sigma[A] \multimap \mathsf{Bool}$ w/o $\&/\oplus \subseteq$ tree-walking automata $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ reg. tree languages

$\Rightarrow$ additive connectives *must* be included to generalize our characterization of regular functions to *trees* (but they are *incompatible* with the aperiodic case)

A big technical digression

Automata over the category $\mathsf{Int}(\mathsf{PFinSet})$ = two-way transducers [Hines 2003]

$\rightsquigarrow$ related to the "geometry of interaction" semantics of linear logic; drawbacks:

- not affine: superficial issue, just use the Interaction Abstract Machine instead (ongoing work with Gabriele Vanoni)
- no additive connectives $\&/\oplus$: a feature rather than a bug, allows proving $\mathsf{Tree}_\Sigma[A] \multimap \mathsf{Bool}$ w/o $\&/\oplus \subseteq$ tree-walking automata $\underbrace{\subsetneq}_{\text{[Bojańczyk \& Colcombet 2005]}}$ reg. tree languages

$\Rightarrow$ additive connectives *must* be included to generalize our characterization of regular functions to *trees* (but they are *incompatible* with the aperiodic case)

Alternatively, in [Gallot, Lemay & Salvati 2020] – work independent from ours

"*Higher-order tree transducer*" whose memory consists of an affine λ-term; no additives, but regular lookaround ($\simeq$ preprocessing on input tree)

Some further developments inspired by “implicit automata”

New automaton/transducer models and/or answers to open problems:

- *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, ...)

Some further developments inspired by "implicit automata"

New automaton/transducer models and/or answers to open problems:

- *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, ...)
- "Collapsible pushdown transducers": answers our running question on STλC
  Lessons from 1980s literature on (higher-order) tree transducers
  + 2010s work on higher-order recursion schemes (+ Plotkin 1982/2022)

Some further developments inspired by “implicit automata”

New automaton/transducer models and/or answers to open problems:

- *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, ...)
- “Collapsible pushdown transducers”: answers our running question on STλC
  Lessons from 1980s literature on (higher-order) tree transducers
  + 2010s work on higher-order recursion schemes (+ Plotkin 1982/2022)
- planar two-way automata (Hines) = star-free languages

Some further developments inspired by "implicit automata"

New automaton/transducer models and/or answers to open problems:

- *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, …)
- "Collapsible pushdown transducers": answers our running question on STλC
  Lessons from 1980s literature on (higher-order) tree transducers
  + 2010s work on higher-order recursion schemes (+ Plotkin 1982/2022)
- planar two-way automata (Hines) = star-free languages
- (Baillot's characterization of P in Elementary Affine Logic) $\setminus$ (recursive types)
  = regular languages my first use of these ideas, [post-proceedings DICE-FOPARA'19]

Some further developments inspired by “implicit automata”

New automaton/transducer models and/or answers to open problems:

- *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, …)
- “Collapsible pushdown transducers”: answers our running question on STλC
  Lessons from 1980s literature on (higher-order) tree transducers
  + 2010s work on higher-order recursion schemes (+ Plotkin 1982/2022)
- planar two-way automata (Hines) = star-free languages
- (Baillot’s characterization of P in Elementary Affine Logic) $\setminus$ (recursive types) = regular languages my first use of these ideas, [post-proceedings DICE-FOPARA’19]
- β-convertibility for the safe λ-calculus is TOWER-complete (new!)

Conclusion

We study the expressive power of typed λ-calculi

$\longrightarrow$ connections with automata theory *naturally* emerge

Characterization of classes of languages using Church encodings

- Regular languages in simply typed λ-calculus [Hillebrand & Kanellakis 1996]
- Star-free languages in non-commutative affine λ-calculus [N. & Pradic 2020]

Many further results on string-to-string (or even tree-to-tree) *functions*:

correspond to *transducers* (automata with output)

Convergence with another tradition coming from automata theory:

higher-order (grammars | tree transducers), recursion schemes, ...

Also a source of inspiration for both λ-calculi and automata

Conclusion

Thanks for your attention! Questions?

We study the expressive power of typed λ-calculi

$\longrightarrow$ connections with automata theory *naturally* emerge

Characterization of classes of languages using Church encodings

- Regular languages in simply typed λ-calculus [Hillebrand & Kanellakis 1996]
- Star-free languages in non-commutative affine λ-calculus [N. & Pradic 2020]

Many further results on string-to-string (or even tree-to-tree) *functions*:

correspond to *transducers* (automata with output)

Convergence with another tradition coming from automata theory:

higher-order (grammars | tree transducers), recursion schemes, ...

Also a source of inspiration for both λ-calculi and automata