Aperiodicity in a non-commutative logic

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

IRIF PhD seminar, March 25th, 2020

Implicit automata in typed λ-calculi: the idea

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

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

Implicit automata in typed λ-calculi: the idea

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

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

Implicit automata in typed λ-calculi: the idea

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

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

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

Implicit automata in typed λ-calculi: the idea

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

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

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

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

Implicit automata in typed λ-calculi: concretely

Template for a theorem

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

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

[1] Hillebrand & Kanellakis 1996

Implicit automata in typed λ-calculi: concretely

Template for a theorem

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

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

[1] Hillebrand & Kanellakis 1996

...

Implicit automata in typed λ-calculi: concretely

Template for a theorem

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

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages
[2]	cyclic proofs (proofs-as-programs)	regular languages

[1] Hillebrand & Kanellakis 1996

...

[2] Kuperberg, Pinault & Pous 2019

Implicit automata in typed λ-calculi: concretely

Template for a theorem

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

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages
[2]	cyclic proofs (proofs-as-programs)	regular languages
[3]	**non-commutative λ-calculus**	**star-free languages**

[1] Hillebrand & Kanellakis 1996

...

[2] Kuperberg, Pinault & Pous 2019

[3] our work! currently submitted

Implicit automata in typed λ-calculi: concretely

Template for a theorem

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

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages
[2]	cyclic proofs (proofs-as-programs)	regular languages
[3]	**non-commutative λ-calculus**	**star-free languages**

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

(explaining the λ-calculus and regular languages along the way)

[2] Kuperberg, Pinault & Pous 2019

[3] our work! currently submitted

Implicit automata in typed λ-calculi: concretely

Template for a theorem

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

	$\mathcal{P}$	$\mathcal{A}$
[1]	simply typed λ-calculus	regular languages
[2]	cyclic proofs (proofs-as-programs)	regular languages
[3]	**non-commutative λ-calculus**	**star-free languages**

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

(explaining the λ-calculus and regular languages along the way)

[2] Kuperberg, Pinault & Pous 2019 (won't be covered)

[3] our work! currently submitted (after [1])

The λ-calculus

A naive syntactic theory of functions:

$$\begin{array}{rcl} f\,x & \approx & f(x) \\ \lambda x.\ t & \approx & x \mapsto t \\ (\lambda x.\ t)\ u =_\beta t\{x := u\} & \approx & (x \mapsto x^2 + 1)(42) = 42^2 + 1 \end{array}$$

The λ-calculus

A naive syntactic theory of functions:

$$
\begin{array}{rcl}
f\,x & \approx & f(x) \\
\lambda x.\ t & \approx & x \mapsto t \\
\lambda x.\ \lambda y.\ t & \approx & x \mapsto (y \mapsto t) \approx (x, y) \mapsto t \\
(\lambda x.\ t)\ u =_\beta t\{x := u\} & \approx & (x \mapsto x^2 + 1)(42) = 42^2 + 1
\end{array}
$$

The λ-calculus

A naive syntactic theory of functions:

$$
\begin{array}{rcl}
f\,x & \approx & f(x) \\
\lambda x.\ t & \approx & x \mapsto t \\
\lambda x.\ \lambda y.\ t & \approx & x \mapsto (y \mapsto t) \approx (x, y) \mapsto t \\
(\lambda x.\ t)\ u =_\beta t\{x := u\} & \approx & (x \mapsto x^2 + 1)(42) = 42^2 + 1
\end{array}
$$

But no primitive data type :-(

The λ-calculus

A naive syntactic theory of functions:

$$\begin{array}{rcl} f\,x & \approx & f(x) \\ \lambda x.\, t & \approx & x \mapsto t \\ \lambda x.\, \lambda y.\, t & \approx & x \mapsto (y \mapsto t) \approx (x, y) \mapsto t \\ (\lambda x.\, t)\; u =_\beta t\{x := u\} & \approx & (x \mapsto x^2 + 1)(42) = 42^2 + 1 \end{array}$$

But no primitive data type :-(

Church encodings of natural numbers:
morally, $n \in \mathbb{N} \rightsquigarrow \overline{n} : (f, x) \mapsto f^n = f \circ \ldots \circ f(x)$

$$\overline{2} = \lambda f x.\, f\,(f\,x) \qquad :\text{-)}$$

The simply typed λ-calculus

Add a *type system*: specifications for λ-terms

$$t : A \to B \quad \approx \quad \text{“}t \text{ is a function from } A \text{ to } B\text{”}$$

Simple types: built from constant o and binary operation ‘$\to$’

$$\dfrac{f : o \to o \qquad \dfrac{f : o \to o \qquad x : o}{f\,x : o}}{f\,(f\,x) : o}$$

The simply typed λ-calculus

Add a *type system*: specifications for λ-terms

$$t : A \to B \quad \approx \quad \text{“}t \text{ is a function from } A \text{ to } B\text{”}$$

Simple types: built from constant o and binary operation ‘$\to$’

$$\dfrac{f : o \to o \qquad \dfrac{f : o \to o \qquad x : o}{f\,x : o}}{f\,(f\,x) : o}$$

$$\overline{2} = \lambda f.\ \lambda x.\ f\,(f\,x) : (o \to o) \to o \to o$$

$\mathsf{Nat} = (o \to o) \to o \to o$ is the type of natural numbers

Defining languages in the simply typed λ-calculus

More Church encodings:

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

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

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

Defining languages in the simply typed λ-calculus

More Church encodings:

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

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

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

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

Defining languages in the simply typed λ-calculus

More Church encodings:

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

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

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

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

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

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

Defining languages in the simply typed λ-calculus

More Church encodings:

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

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

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

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

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

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

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

Defining languages in the simply typed λ-calculus

More Church encodings:

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

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

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

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

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

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

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

Theorem (Hillebrand & Kanellakis 1996)

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

Regular languages

Many equivalent definitions:

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

Regular languages

Many equivalent definitions:

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

Theorem (classical)

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

Σ: finite alphabet, Σ^*: words over Σ

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

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

Regular languages

Many equivalent definitions:

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

Theorem (classical)

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

Σ: finite alphabet, Σ^*: words over Σ

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

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

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star* (except Σ^*)

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

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star* (except Σ^*)

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

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

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star* (except Σ^*)

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

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

Theorem (Schützenberger 1965)

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

Star-free languages and aperiodicity

Star-free languages: regular expressions with *complementation* but *without repetition star* (except Σ^*)

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

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

Theorem (Schützenberger 1965)

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

Definition

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

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

From aperiodicity to non-commutativity

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

From aperiodicity to non-commutativity

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

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

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

From aperiodicity to non-commutativity

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

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

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

Idea: make the order of arguments matter

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

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

Non-commutative types and linear logic

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

Non-commutative types and linear logic

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

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

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

Non-commutative types and linear logic

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

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

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

Fix: prohibit duplication, as in linear logic (or C++ `unique_ptr`)

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

Non-commutative types and linear logic

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

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

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

Fix: prohibit duplication, as in linear logic (or C++ `unique_ptr`)

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

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

Finally, our theorem

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

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

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

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

Finally, our theorem

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

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

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

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

Theorem (N. & Pradic 2020)

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

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

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

Next, some technical details:

- proving (one direction of) Hillebrand–Kanellakis: λ-definable $\subseteq$ regular, by semantic evaluation
- a few words on the proof of our theorem
- rules of our type system

Set-theoretic semantics of the simply typed λ-calculus

Canonical semantics: formalize the idea of λ-terms as functions

- choose set S, $[\![o]\!] = S$, $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$
- $t : A \rightsquigarrow [\![t]\!] \in [\![A]\!]$ by induction, e.g. $[\![f\ x]\!] = [\![f]\!]\,([\![x]\!])$
- soundness: $t =_\beta u \implies [\![t]\!] = [\![u]\!]$

Set-theoretic semantics of the simply typed λ-calculus

Canonical semantics: formalize the idea of λ-terms as functions

- choose set S, $[\![o]\!] = S$, $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$
- $t : A \rightsquigarrow [\![t]\!] \in [\![A]\!]$ by induction, e.g. $[\![f\ x]\!] = [\![f]\!]\,([\![x]\!])$
- soundness: $t =_\beta u \implies [\![t]\!] = [\![u]\!]$

Lemma

If $[\![o]\!]$ is finite, then $[\![A]\!]$ is finite for any type A.

Set-theoretic semantics of the simply typed λ-calculus

Canonical semantics: formalize the idea of λ-terms as functions

- choose set S, $[\![o]\!] = S$, $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$
- $t : A \rightsquigarrow [\![t]\!] \in [\![A]\!]$ by induction, e.g. $[\![f\ x]\!] = [\![f]\!]\,([\![x]\!])$
- soundness: $t =_\beta u \implies [\![t]\!] = [\![u]\!]$

Lemma

If $[\![o]\!]$ is finite, then $[\![A]\!]$ is finite for any type A.

Lemma

$[\![\overline{w\mathtt{0}}]\!]$ *and* $[\![\overline{w\mathtt{1}}]\!]$ *depend only on* $[\![\overline{w}]\!]$.

Proof: $\mathsf{append0}$ definable so that $\mathsf{append0}\ \overline{w} =_\beta \overline{w\mathtt{0}}$

$[\![\overline{w\mathtt{0}}]\!] = [\![\mathsf{append0}\ \overline{w}]\!] = [\![\mathsf{append0}]\!]\,([\![w]\!])$

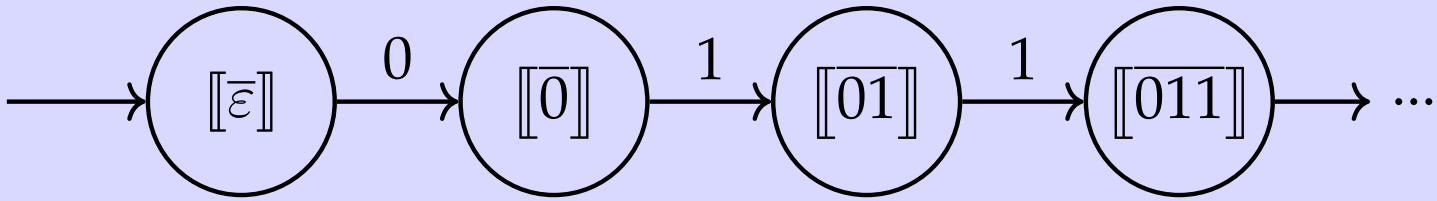
λ-definable languages are regular

Theorem (Hillebrand & Kanellakis, LICS'96)

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

Proof.

We build a DFA with *finite* set of states $Q = [\![\mathsf{Str}_{\{0,1\}}[A]]\!]$ ($\mathrm{Card}(Q)$ depends on A), acceptation as $[\![t]\!](-) = [\![\mathtt{true}]\!]$.



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

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

Remarks on the non-commutative case

That was the "hard" inclusion: λ-definable $\subseteq$ regular.

This methods adapts (with a few subtleties) to our case:

Lemma (in our non-commutative λ-calculus)

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

Remarks on the non-commutative case

That was the "hard" inclusion: λ-definable $\subseteq$ regular.

This methods adapts (with a few subtleties) to our case:

Lemma (in our non-commutative λ-calculus)

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

But the star-free $\subseteq$ λ-definable direction is harder:

how do we exploit the aperiodicity assumption?

Remarks on the non-commutative case

That was the "hard" inclusion: λ-definable $\subseteq$ regular.
This methods adapts (with a few subtleties) to our case:

Lemma (in our non-commutative λ-calculus)

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

But the star-free $\subseteq$ λ-definable direction is harder:
how do we exploit the aperiodicity assumption?

Theorem (Krohn & Rhodes 1965)

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

Convenient version for us: *aperiodic sequential transducers* = function comp. of transducers w/ *2 states* $\rightsquigarrow \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$

A non-commutative affine type system

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

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

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

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

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

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

Conclusion / further work

Successful revival of ideas from the 1990s:
automata vs λ-calculus, non-commutative type systems

Next: string-to-string functions in typed λ-calculi
so-called *copyless* streaming transducers $\cong$ affine typing

Conclusion / further work

Successful revival of ideas from the 1990s:
automata vs λ-calculus, non-commutative type systems

Next: string-to-string functions in typed λ-calculi
so-called *copyless* streaming transducers $\cong$ affine typing

- regular functions $\Gamma^* \to \Sigma^*$ (MSO transductions):
  $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ with commutative affine types

Conclusion / further work

Successful revival of ideas from the 1990s:
automata vs λ-calculus, non-commutative type systems

Next: string-to-string functions in typed λ-calculi
so-called *copyless* streaming transducers $\cong$ affine typing

- regular functions $\Gamma^* \to \Sigma^*$ (MSO transductions):
  $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ with commutative affine types
- *aperiodic* regular functions (FO transductions):
  $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}[o]$ with non-commutativity

Conclusion / further work

Successful revival of ideas from the 1990s:
automata vs λ-calculus, non-commutative type systems

Next: string-to-string functions in typed λ-calculi
so-called *copyless* streaming transducers $\cong$ affine typing

- regular functions $\Gamma^* \to \Sigma^*$ (MSO transductions):
  $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ with commutative affine types
- *aperiodic* regular functions (FO transductions):
  $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}[o]$ with non-commutativity
- proofs via monoidal closed categories, justifies the complicated definition of streaming *tree* transducers

Conclusion / further work

Successful revival of ideas from the 1990s:
automata vs λ-calculus, non-commutative type systems

Next: string-to-string functions in typed λ-calculi
so-called *copyless* streaming transducers $\cong$ affine typing

- regular functions $\Gamma^* \to \Sigma^*$ (MSO transductions):
  $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma[o]$ with commutative affine types
- *aperiodic* regular functions (FO transductions):
  $\mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}[o]$ with non-commutativity
- proofs via monoidal closed categories, justifies the complicated definition of streaming *tree* transducers
- in the simply typed λ-calculus: open problem!