Implicit automata in typed λ-calculi

Lê Thành Dũng (Tito) Nguyễn — `nltd@nguyentito.eu` — ÉNS Lyon
partially based on joint work with Cécilia Pradic (Swansea University)

Università di Bologna, April 5th, 2023

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 for a long time, 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*.

Simply typed functions on Church numerals (1)

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

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

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

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 $\mathsf{Nat} = (o \to o) \to o \to o$

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

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$ therefore
  $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

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

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 (1)

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

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

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$ therefore
  $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

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

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 (1)

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

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

- for $n \in \mathbb{N}$, we have $\overline{n} : \mathsf{Nat}$ therefore
  $\overline{n} : \mathsf{Nat}[A] = \mathsf{Nat}[A/o] = (A \to A) \to A \to A$
- conversely $t : \mathsf{Nat} \implies \exists n \in \mathbb{N}.\ t =_{\beta\eta} \overline{n}$

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

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?

For $\mathsf{Nat}[A] \to \mathsf{Bool}$, there is a nice characterisation!

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

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

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

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

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

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

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

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

Example: $t = \lambda s.\, s\ \texttt{id}\ \texttt{not}\ \texttt{true} : \mathsf{Str}_{\{0,1\}}[\mathsf{Bool}] \to \mathsf{Bool}$ (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 STLC 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 $\mathcal{C}$.

(dichotomy taken from Abramsky, cf. e.g. the *Structure meets Power* workshop)

- 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 STLC 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 $\mathcal{C}$.

(dichotomy taken from Abramsky, cf. e.g. the *Structure meets Power* workshop)

- 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 STLC 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 $\mathcal{C}$.

(dichotomy taken from Abramsky, cf. e.g. the *Structure meets Power* workshop)

- 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*

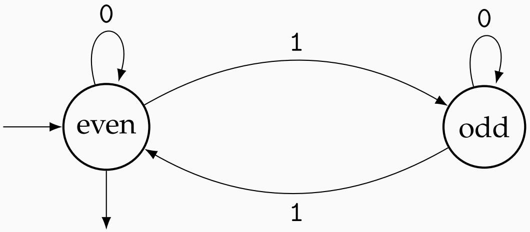
Yet *we didn't ask* for regular languages to appear in the simply typed λ-calculus!

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

Regular languages

Many classical equivalent definitions (+ STLC 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 (+ STLC 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* $L = \varphi^{-1}(P) = \{w \in \Sigma^* \mid \varphi(w) \in P\}$.

Regular languages

Many classical equivalent definitions (+ STLC 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* $L = \varphi^{-1}(P) = \{w \in \Sigma^* \mid \varphi(w) \in P\}$.

Later in the talk: generalizations to *functions* $\Sigma^* \to \Gamma^*$

Before that: proof for languages in STLC

Proof of STLC-definable $\implies$ regular

Theorem (Hillebrand & Kanellakis, LICS'96)

For any type A and any simply typed λ-term $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$, the language $\mathcal{L}(t) = \{w \in \Sigma^* \mid t\,\overline{w} \to_\beta^* \texttt{true}\}$ 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:

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

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

Proof of STLC-definable $\implies$ regular

Theorem (Hillebrand & Kanellakis, LICS'96)

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

Part 2 of proof.

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

$$\begin{tikzcd} {} \arrow[r] & {[\![\overline{\varepsilon}]\!]} \arrow[r, "\texttt{a}"] & {[\![\overline{\texttt{a}}]\!]} \arrow[r, "\texttt{b}"] & {[\![\overline{\texttt{ab}}]\!]} \arrow[r, "\texttt{b}"] & {[\![\overline{\texttt{abb}}]\!]} \arrow[r] & \cdots \end{tikzcd}$$

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

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

Proof of STLC-definable $\implies$ regular

Theorem (Hillebrand & Kanellakis, LICS'96)

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

Part 2 of proof.

We use $[\![-]\!] : \mathrm{STLC} \to \mathsf{FinSet}$ 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 $[\![\mathsf{Str}_\Sigma[A]]\!]$) $\square$

Usual transduction classes (functions $\Gamma^* \to \Sigma^*$)

What is the right generalization of regular languages to *transductions*?

There are several canonical ones!

$$\underbrace{\text{sequential fonctions}}_{\textit{deterministic}\text{ finite transducers}} \subsetneq \underbrace{\text{rational functions}}_{\textit{nondeterministic}\text{ transducers}} \subsetneq \text{regular functions}$$

(“rational languages” = name for regular languages in the French tradition)

Usual transduction classes (functions $\Gamma^* \to \Sigma^*$)

What is the right generalization of regular languages to *transductions*?

There are several canonical ones!

$$\underbrace{\text{sequential fonctions}}_{\textit{deterministic}\text{ finite transducers}} \subsetneq \underbrace{\text{rational functions}}_{\textit{nondeterministic}\text{ transducers}} \subsetneq \text{regular functions}$$

(“rational languages” = name for regular languages in the French tradition)

- those classes are *closed under composition*
- preservation property: L regular $\implies f^{-1}(L)$ regular

Usual transduction classes (functions $\Gamma^* \to \Sigma^*$)

What is the right generalization of regular languages to *transductions*?

There are several canonical ones!

$$\underbrace{\text{sequential fonctions}}_{\textit{deterministic}\text{ finite transducers}} \subsetneq \underbrace{\text{rational functions}}_{\textit{nondeterministic}\text{ transducers}} \subsetneq \text{regular functions}$$

("rational languages" = name for regular languages in the French tradition)

- those classes are *closed under composition*
- preservation property: L regular $\implies f^{-1}(L)$ regular

We shall be interested in *regular functions*; many definitions,

such as *streaming string transducers*

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

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

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

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

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

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:

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

$\downarrow$

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

$X = ca \qquad Y = \varepsilon$

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

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

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

		↓							
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:

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

$\downarrow$

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

Linearity

Regular functions = computed by *copyless* streaming string transducers

Restrictions on "copying power": old theme in automaton theory

Linearity

Regular functions = computed by *copyless* streaming string transducers

Restrictions on "copying power": old theme in automaton theory

λ-calculus counterpart: linear types (Girard 1987)

Here, we use the "$\lambda\ell^{\oplus\&}$-calculus" = *Dual Intuitionistic Linear Logic*
+ additive connectives $\oplus, \&$

(linear = *exactly* once; additives allow us to simulate *at most* once (*affine*))

Linearity

Regular functions = computed by *copyless* streaming string transducers

Restrictions on "copying power": old theme in automaton theory

λ-calculus counterpart: <u>linear</u> types (Girard 1987)

Here, we use the "$\lambda\ell^{\oplus\&}$-calculus" = *Dual Intuitionistic Linear Logic* + additive connectives $\oplus, \&$

(linear = *exactly* once; additives allow us to simulate *at most* once (*affine*))

$$A, B ::= o \mid \overbrace{\underbrace{A \multimap B}_{\text{linear functions}} \mid A \otimes B}^{\text{multiplicatives}} \mid \overbrace{A \mathbin{\&} B \mid A \oplus B}^{\text{additives}} \mid \underbrace{A \to B}_{\text{non-linear functions}}$$

First steps in implicit transducers

Linear Church encodings

$\mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to (o \multimap o)$, *mutatis mutandis* for Str_Σ

Definition: a type of the $\lambda\ell^{\oplus\&}$-calculus is *purely linear* if it contains no "$\to$"

Theorem (N. & Pradic; proof technique on next slides)

$f : \Gamma^* \to \Sigma^*$ *is* regular $\iff$ $\exists$ *a* purely linear *type* A *and* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *in the* $\lambda\ell^{\oplus\&}$*-calculus such that* $\forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$

First steps in implicit transducers

Linear Church encodings

$\mathsf{Str}_{\{a,b\}} = (o \multimap o) \to (o \multimap o) \to (o \multimap o)$, *mutatis mutandis* for Str_Σ

Definition: a type of the $\lambda\ell^{\oplus\&}$-calculus is *purely linear* if it contains no "$\to$"

Theorem (N. & Pradic; proof technique on next slides)

$f : \Gamma^* \to \Sigma^*$ *is* regular $\iff$ $\exists$ *a* purely linear *type* A *and* $t : \mathsf{Str}_\Gamma[A] \multimap \mathsf{Str}_\Sigma$ *in the* $\lambda\ell^{\oplus\&}$*-calculus such that* $\forall w \in \Gamma^*,\ \overline{f(w)} =_\beta t\ \overline{w}$

- works also for regular tree-to-tree functions
  - similar to "linear high-order tree transducers" [Gallot, Lemay & Salvati 2020] $\to$ characterize reg. tree fn. w/o additives, but with "regular lookahead"
- $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$ instead $\rightsquigarrow$ *discovery* [N., Noûs & Pradic 2021] of a natural subclass of the *polyregular functions* [Bojańczyk 2018]

Semantic evaluation and categorical automata theory

Again, go from typed λ-terms to automata via *denotational semantics*

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

Semantic evaluation and categorical automata theory

Again, go from typed λ-terms to automata via *denotational semantics*

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

Approach for our "implicit transducer" results

- Find a *monoidal closed category* $\mathcal{C}$
  (provides a semantics for the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus)

Semantic evaluation and categorical automata theory

Again, go from typed λ-terms to automata via *denotational semantics*

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

Approach for our "implicit transducer" results

- Find a *monoidal closed category* $\mathcal{C}$
  (provides a semantics for the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus)
- such that $\mathcal{C}$*-automata* compute regular functions
  (notion of automaton over a category used here: [Colcombet & Petrişan 2017])

Semantic evaluation and categorical automata theory

Again, go from typed λ-terms to automata via *denotational semantics*

- naive semantics of the simply typed λ-calculus in *finite sets*
  $\longrightarrow$ finite automata (Hillebrand & Kanellakis's theorem)

Approach for our "implicit transducer" results

- Find a *monoidal closed category* $\mathcal{C}$
  (provides a semantics for the purely linear fragment of the $\lambda\ell^{\oplus\&}$-calculus)
- such that *$\mathcal{C}$-automata* compute regular functions
  (notion of automaton over a category used here: [Colcombet & Petrişan 2017])

e.g. $\mathcal{C} = \mathsf{Int}(\mathsf{PFinSet}) \rightsquigarrow$ two-way automata [Hines 2003]
$\longrightarrow$ connection with geometry of interaction, but does not handle additives

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_\oplus$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_\oplus$ = free finite coproduct completion $\simeq$ adds finite states

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\underline{\mathcal{SR}_\oplus\text{-automata}}$

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_\oplus$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_\oplus$ is merely "partially" monoidal closed...

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_\oplus$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_\oplus$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_\oplus$ is merely "partially" monoidal closed...
- However, $(\mathcal{SR}_\&)_\oplus$ is monoidal closed ($(-)_\&$ = product completion)

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\underline{\mathcal{SR}_{\oplus}\text{-automata}}$

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_{\oplus}$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_{\oplus}$ is merely "partially" monoidal closed...
- However, $(\mathcal{SR}_{\&})_{\oplus}$ is monoidal closed ($(-)_{\&}$ = product completion)

$((-)_{\&})_{\oplus}$ completion $\simeq$ adding finite states with non-determinism

- copyless SSTs can be determinised [Alur & Deshmukh 2011]

Monoidal closed categories vs streaming string transducers

Copyless streaming string transducers $\simeq$ $\mathcal{SR}_{\oplus}$-automata

- $\mathcal{SR}$ = category of copyless transitions between finite sets of registers
- $(-)_{\oplus}$ = free finite coproduct completion $\simeq$ adds finite states

- Issue: $\mathcal{SR}_{\oplus}$ is merely "partially" monoidal closed…
- However, $(\mathcal{SR}_{\&})_{\oplus}$ is monoidal closed ($(-)_{\&}$ = product completion)

$((-)_{\&})_{\oplus}$ completion $\simeq$ adding finite states with non-determinism

- copyless SSTs can be determinised [Alur & Deshmukh 2011]
- **our work:** *categorical* determinisation using the existence of some "function spaces" $A \multimap B$ = "partial" monoidal closure!

Monoidal closed structure on the category $(\mathcal{SR}_{\&})_{\oplus}$

For $A, B \in \mathsf{Obj}(\mathcal{SR})$: decompose morphisms

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \quad \rightsquigarrow \quad \text{shape} \begin{cases} X := Z_1 X Z_2 Y \\ Y := Z_3 \end{cases} \quad + \quad \text{labels } Z_1 = ab, \ldots$$

$$\textit{copyless} \implies \textit{finite} \text{ number of possible shapes} \implies A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$$

Monoidal closed structure on the category $(\mathcal{SR}_{\&})_{\oplus}$

For $A, B \in \mathsf{Obj}(\mathcal{SR})$: decompose morphisms

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \quad \rightsquigarrow \quad \text{shape} \begin{cases} X := Z_1 X Z_2 Y \\ Y := Z_3 \end{cases} \quad + \quad \text{labels } Z_1 = ab, \ldots$$

$$\textit{copyless} \implies \textit{finite} \text{ number of possible shapes} \implies A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$$

This decomposition is a classical technique, also used to show that comparison-free polyregular functions are closed under composition in our ICALP'21 paper.

Monoidal closed structure on the category $(\mathcal{SR}_{\&})_{\oplus}$

In $(\mathcal{C}_{\&})_{\oplus}$, we have $\bigoplus_u \&_x A_{u,x} \multimap \bigoplus_v \&_y B_{v,y} \;=\; \&_u \bigoplus_v \&_y \bigoplus_x A_{u,x} \multimap B_{v,y}$

Theorem ("Dialectica-like" construction inspired by [Gödel, de Paiva, Hofstra])

Let $\mathcal{C}$ a symmetric monoidal category. If the function space $A \multimap B$ exists in $\mathcal{C}_{\oplus}$ for all $A, B \in \mathsf{Obj}(\mathcal{C})$, then $(\mathcal{C}_{\&})_{\oplus}$ is monoidal closed.

For $A, B \in \mathsf{Obj}(\mathcal{SR})$: decompose morphisms

$$\begin{cases} X := abXcY \\ Y := ba \end{cases} \quad \rightsquigarrow \quad \text{shape} \begin{cases} X := Z_1 X Z_2 Y \\ Y := Z_3 \end{cases} \quad + \quad \text{labels } Z_1 = ab, \ldots$$

$$\textit{copyless} \implies \textit{finite} \text{ number of possible shapes} \implies A \multimap B \in \mathsf{Obj}(\mathcal{SR}_{\oplus})$$

This decomposition is a classical technique, also used to show that comparison-free polyregular functions are closed under composition in our ICALP'21 paper.

What happens without linearity?

$$\begin{aligned}\text{Copyless streaming string transducers} &\iff \text{regular functions}\\ &\iff \text{a linear } \lambda\text{-calculus } (\lambda\ell^{\oplus\&}).\end{aligned}$$

What happens without linearity?

$$\begin{aligned}\text{Copyless streaming string transducers} &\iff \text{regular functions}\\ &\iff \text{a linear } \lambda\text{-calculus } (\lambda\ell^{\oplus\&}).\end{aligned}$$

Let's drop linearity: *copyful* SSTs can be encoded in the simply typed λ-calculus.

- polynomial example: $abc \mapsto (a)(ab)(abc)$ with $a \mapsto \begin{cases} X := Xa \\ Y := YX \end{cases}$

What happens without linearity?

$$\begin{aligned}\text{Copyless streaming string transducers} &\iff \text{regular functions}\\ &\iff \text{a linear } \lambda\text{-calculus } (\lambda\ell^{\oplus\&}).\end{aligned}$$

Let's drop linearity: *copyful* SSTs can be encoded in the simply typed λ-calculus.

- polynomial example: $abc \mapsto (a)(ab)(abc)$ with $a \mapsto \begin{cases} X := Xa \\ Y := YX \end{cases}$
- can grow up to exponentially, e.g. $X := XX$

What happens without linearity?

Copyless streaming string transducers $\iff$ regular functions

$\iff$ a linear λ-calculus ($\lambda\ell^{\oplus\&}$).

Let's drop linearity: *copyful* SSTs can be encoded in the simply typed λ-calculus.

- polynomial example: $abc \mapsto (a)(ab)(abc)$ with $a \mapsto \begin{cases} X := Xa \\ Y := YX \end{cases}$
- can grow up to exponentially, e.g. $X := XX$

$\longrightarrow$ not closed under composition; by composing we get towers of exp, matching the known growth rate for simply typed λ-calculus

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

What happens without linearity?

$$\begin{aligned}\text{Copyless streaming string transducers} &\iff \text{regular functions}\\ &\iff \text{a linear } \lambda\text{-calculus } (\lambda\ell^{\oplus\&}).\end{aligned}$$

Let's drop linearity: *copyful* SSTs can be encoded in the simply typed λ-calculus.

- polynomial example: $abc \mapsto (a)(ab)(abc)$ with $a \mapsto \begin{cases} X := Xa \\ Y := YX \end{cases}$
- can grow up to exponentially, e.g. $X := XX$

$\longrightarrow$ not closed under composition; by composing we get towers of exp, matching the known growth rate for simply typed λ-calculus

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

So, what is known about (compositions of) copyful SSTs?

A robust class of hyperexponential transductions

Macro tree transducers (MTTs) generalize copyful SSTs to trees.

Theorem (Engelfriet & Vogler 1988)

Compositions of MTTs $\iff$ *iterated pushdown transducers*

using stacks of … of stacks of input pointers

$\iff$ *"High level tree transducers"*

$\simeq$ *registers storing* simply typed λ-terms

Trivial observation

This is included in the simply typed λ-definable tree functions.

A robust class of hyperexponential transductions

Macro tree transducers (MTTs) generalize copyful SSTs to trees.

Theorem (Engelfriet & Vogler 1988)

Compositions of MTTs $\iff$ *iterated pushdown transducers*

using stacks of … of stacks of input pointers

$\iff$ *"High level tree transducers"*

$\simeq$ *registers storing* simply typed λ-terms

Trivial observation

This is included in the simply typed λ-definable tree functions.

But we'll see why the converse might fail, via a detour through *infinite* structures

(subtle restrictions on which λ-terms may appear in registers)

Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input

$$(q_0, [])$$

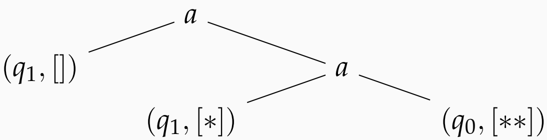
Higher-order pushdown automata = iterated pushdown transducers without input

$$a$$

$$(q_1, [\,]) \qquad\qquad (q_0, [*])$$

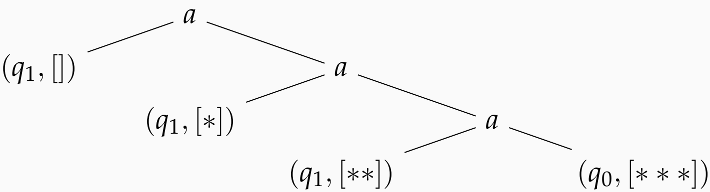
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



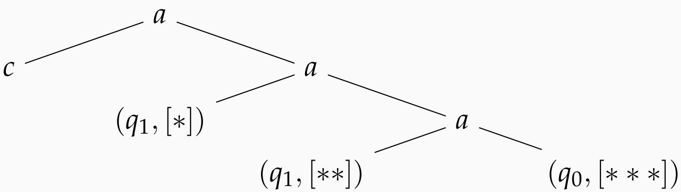
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



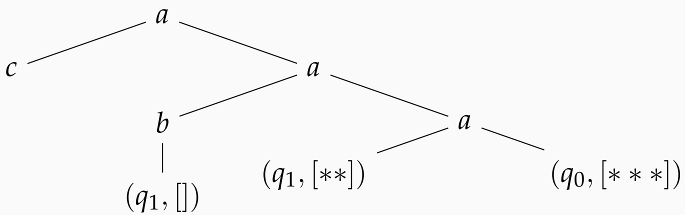
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



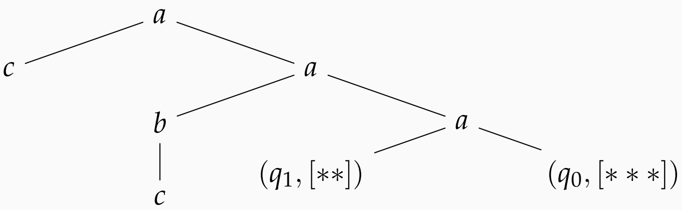
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



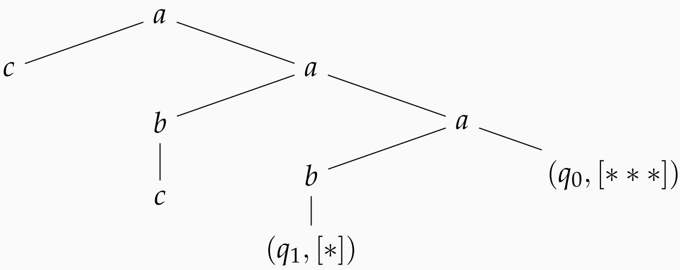
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



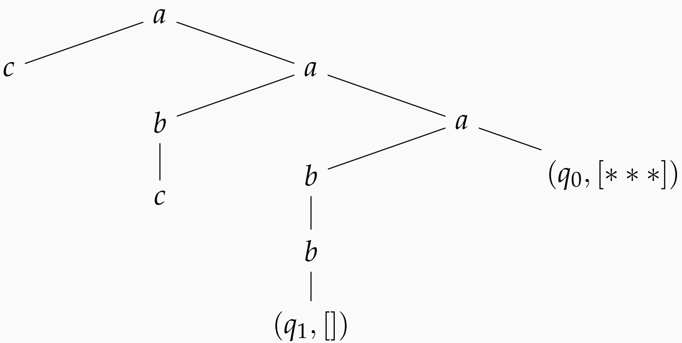
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



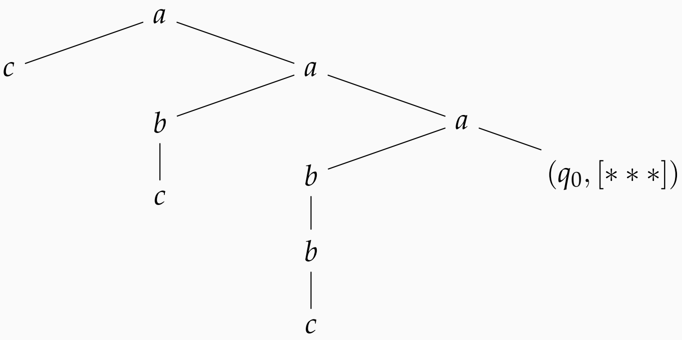
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



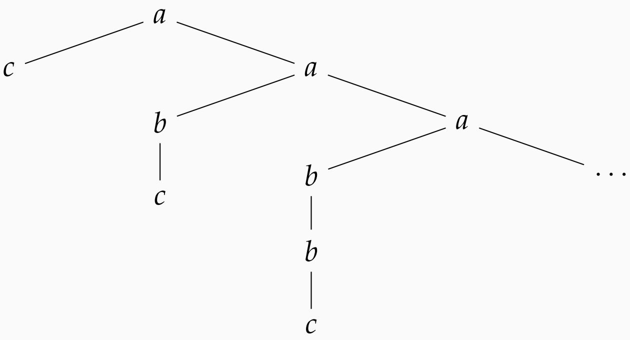
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



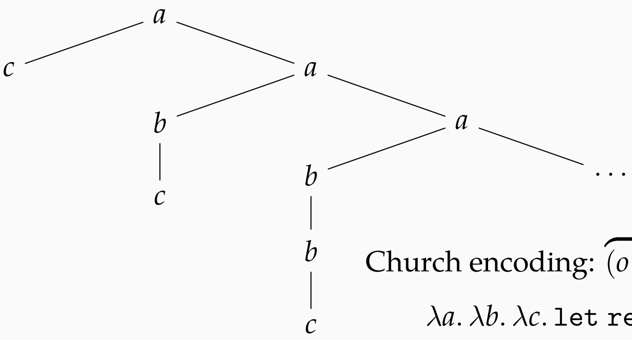
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



Generating infinite trees

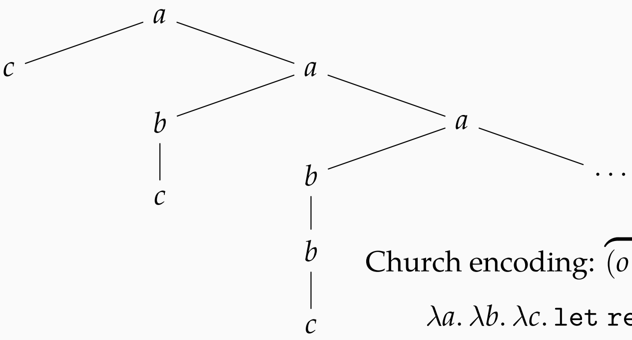
Higher-order pushdown automata = iterated pushdown transducers without input



Church encoding: $\overbrace{(o \to o \to o)}^{a} \to \overbrace{(o \to o)}^{b} \to \overset{\overset{c}{\downarrow}}{o} \to o$

$$\lambda a.\, \lambda b.\, \lambda c.\, \texttt{let rec}\, f = \lambda x.\, a\, x\, (f\, (b\, x))\ \texttt{in}\, f\, c$$

Higher-order pushdown automata = iterated pushdown transducers without input



Church encoding: $\overbrace{(o \to o \to o)}^{a} \to \overbrace{(o \to o)}^{b} \to \overset{\overset{c}{\downarrow}}{o} \to o$

$$\lambda a.\, \lambda b.\, \lambda c.\, \texttt{let rec}\, f = \lambda x.\, a\, x\, (f\, (b\, x))\ \texttt{in}\, f\, c$$

Theorem (Damm '82; Knapkik, Niwiński & Urzyczyn '02; Salvati & Walukiewicz '12)

HOPDA $\iff$ *so-called* safe *fragment of the simply typed λ-calculus with* `let rec`

Safely λ-definable functions

Equivalence for formalisms generating infinite trees

Higher-order pushdown automata $\iff$ *safe* λ-calculus with `let rec`

- Engelfriet & Vogler's "high level tree transducers" are directly inspired from Damm's work on higher-order grammars $\rightarrow$ implicit safety constraint

Safely λ-definable functions

Equivalence for formalisms generating infinite trees

Higher-order pushdown automata $\iff$ *safe* λ-calculus with `let rec`

- Engelfriet & Vogler's "high level tree transducers" are directly inspired from Damm's work on higher-order grammars $\rightarrow$ implicit safety constraint

$\rightarrow$ Claim: the following should follow mostly routinely from previous work

Safe λ-terms (w/o `let rec` [Blum & Ong 2009]) of type $\mathtt{Tree}_\Gamma[A] \rightarrow \mathtt{Tree}_\Sigma$ compute the same functions as "high level TTs" / ...

Safely λ-definable functions

Equivalence for formalisms generating infinite trees

Higher-order pushdown automata $\iff$ *safe* λ-calculus with `let rec`

- Engelfriet & Vogler's "high level tree transducers" are directly inspired from Damm's work on higher-order grammars $\rightarrow$ implicit safety constraint

$\rightarrow$ <u>Claim:</u> the following should follow mostly routinely from previous work

Safe λ-terms (w/o `let rec` [Blum & Ong 2009]) of type $\mathtt{Tree}_\Gamma[A] \rightarrow \mathtt{Tree}_\Sigma$ compute the same functions as "high level TTs" / ...

But some trees can only be generated by *unsafe* recursion schemes [Parys 2012]
$\longrightarrow$ safety could also decrease the λ-definable functions on finite trees

Collapsible pushdown transducers

Theorem (Hague, Murawski, Ong & Serre 2008)

Collapsible PDA *generate the same trees as simply typed λ-terms with* `let rec`

Additional structure on pushdowns of ... of pushdowns + collapse operation

Collapsible pushdown transducers

Theorem (Hague, Murawski, Ong & Serre 2008)

Collapsible PDA *generate the same trees as simply typed λ-terms with* `let rec`

Additional structure on pushdowns of ... of pushdowns + collapse operation

The "obvious" theorem

The simply typed λ-definable functions (over Church encodings) are exactly those computable by some "collapsible pushdown tree transducer" model.

Collapsible pushdown transducers

Theorem (Hague, Murawski, Ong & Serre 2008)

Collapsible PDA *generate the same trees as simply typed λ-terms with* `let rec`

Additional structure on pushdowns of … of pushdowns + collapse operation

The "obvious" theorem

The simply typed λ-definable functions (over Church encodings) are exactly those computable by some "collapsible pushdown tree transducer" model.

- Engelfriet & Vogler's proofs rely on inductive characterizations that are not available anymore in this setting…
- Technical issue: "collapsible pushdown transducers" can loop forever,
  the simply typed λ-calculus is terminating

Decomposing the “obvious” theorem: taking divergence into account

Let f : {finite trees} $\rightharpoonup$ {possibly infinite trees} be a partial function.

1. f is computed by a collapsible pushdown transducer
   $\iff$ f is defined by a simply typed λ-term with `let rec`
   $\rightsquigarrow$ straightforward variant of existing proof [Salvati & Walukiewicz 2012]

Decomposing the “obvious” theorem: taking divergence into account

Let $f : \{\text{finite trees}\} \rightharpoonup \{\text{possibly infinite trees}\}$ be a partial function.

1. f is computed by a collapsible pushdown transducer
   $\iff$ f is defined by a simply typed λ-term with `let rec`
   $\rightsquigarrow$ straightforward variant of existing proof [Salvati & Walukiewicz 2012]
2. Furthermore, in that case, there is a simply typed λ-term *without* `let rec` defining a function that coincides with f on $f^{-1}(\{\text{finite trees}\})$
   $\rightsquigarrow$ Plotkin, *Recursion does not always help*, 1982 – arXived in 2022
   again a finite semantics argument! (domain theory)

Conclusion

- "Implicit" characterisations of function classes defined by automata,
  answering *natural* questions about (simply or linearly) typed λ-calculi
- Semantic evaluation technique $\to$ application of denotational semantics
  $+$ connections with categorical approaches to automata
- Discovery of an interesting class of string-to-string functions
  $\longrightarrow$ how I left linear logic and became a transducer theorist `;-)`

Conclusion

- "Implicit" characterisations of function classes defined by automata,
  answering *natural* questions about (simply or linearly) typed λ-calculi
- Semantic evaluation technique $\to$ application of denotational semantics
  + connections with categorical approaches to automata
- Discovery of an interesting class of string-to-string functions
  $\longrightarrow$ how I left linear logic and became a transducer theorist `;-)`

Not mentioned:

- Non-commutative λ-calculus and star-free languages [N. & Pradic, ICALP'20]
- Safe λ-calculus normalization is Tower-complete, via star-free expressions
- Applications of the geometry of interaction (ongoing)

Conclusion

- "Implicit" characterisations of function classes defined by automata,
  answering *natural* questions about (simply or linearly) typed λ-calculi
- Semantic evaluation technique $\to$ application of denotational semantics
  + connections with categorical approaches to automata
- Discovery of an interesting class of string-to-string functions
  $\longrightarrow$ how I left linear logic and became a transducer theorist ;-)

Not mentioned:

- Non-commutative λ-calculus and star-free languages [N. & Pradic, ICALP'20]
- Safe λ-calculus normalization is TOWER-complete, via star-free expressions
- Applications of the geometry of interaction (ongoing)

Take-home theorem [Hillebrand & Kanellakis 1996]

$\mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ in simply typed λ-calculus = regular languages

Conclusion

- "Implicit" characterisations of function classes defined by automata, answering *natural* questions about (simply or linearly) typed λ-calculi
- Semantic evaluation technique $\to$ application of denotational semantics + connections with categorical approaches to automata
- Discovery of an interesting class of string-to-string functions $\longrightarrow$ how I left linear logic and became a transducer theorist ;-)

Not mentioned:

- Non-commutative λ-calculus and star-free languages [N. & Pradic, ICALP'20]
- Safe λ-calculus normalization is Tower-complete, via star-free expressions
- Applications of the geometry of interaction (ongoing)

Take-home theorem [Hillebrand & Kanellakis 1996]

$\mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ in simply typed λ-calculus = regular languages

Conclusion

- "Implicit" characterisations of function classes defined by automata, answering *natural* questions about (simply or linearly) typed λ-calculi
- Semantic evaluation technique $\to$ application of denotational semantics + connections with categorical approaches to automata
- Discovery of an interesting class of string-to-string functions $\longrightarrow$ how I left linear logic and became a transducer theorist ;-)

Not mentioned:

- Non-commutative λ-calculus and star-free languages [N. & Pradic, ICALP'20]
- Safe λ-calculus normalization is Tower-complete, via star-free expressions
- Applications of the geometry of interaction (ongoing)

Take-home theorem [Hillebrand & Kanellakis 1996]

$\mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ in simply typed λ-calculus = regular languages

Conclusion

Thanks for your attention!

- "Implicit" characterisations of function classes defined by automata, answering *natural* questions about (simply or linearly) typed λ-calculi
- Semantic evaluation technique $\to$ application of denotational semantics + connections with categorical approaches to automata
- Discovery of an interesting class of string-to-string functions $\longrightarrow$ how I left linear logic and became a transducer theorist ;-)

Not mentioned:

- Non-commutative λ-calculus and star-free languages [N. & Pradic, ICALP'20]
- Safe λ-calculus normalization is Tower-complete, via star-free expressions
- Applications of the geometry of interaction (ongoing)

Take-home theorem [Hillebrand & Kanellakis 1996]

$\mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ in simply typed λ-calculus = regular languages