A transducer model for simply typed λ-definability

Lê Thành Dũng (Tito) Nguyễn – inspired by previous joint work with Cécilia Pradic

Updated version of a talk given in 2022 (Marseille, Warszawa, Lyon)

The big picture

Basic motivation: natural questions about the *expressiveness of typed λ-calculi*

(minimalistic functional programming languages)

which seem to be related to *finite-state computation*

The big picture

Basic motivation: natural questions about the *expressiveness of typed λ-calculi*

(minimalistic functional programming languages)

which seem to be related to *finite-state computation*

What's in my PhD thesis (j.w.w. Cécilia Pradic)

Connections between type systems inspired by linear logic and contemporary automata/transducer theory (e.g. (poly)regular functions)

The big picture

Basic motivation: natural questions about the *expressiveness of typed λ-calculi*

(minimalistic functional programming languages)

which seem to be related to *finite-state computation*

What's in my PhD thesis (j.w.w. Cécilia Pradic)

Connections between type systems inspired by linear logic and contemporary automata/transducer theory (e.g. (poly)regular functions)

New result

Answer an old open problem on the λ-calculus, taking inspiration from

- a bunch of (sometimes old) transducer models $\to$ covered in the talk
- more recent work on higher-order recursion schemes

+ raise some speculative questions in pure automata theory

The λ-calculus and Church encodings

A naive syntactic theory of functions:

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

The λ-calculus and Church encodings

A naive syntactic theory of functions:

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

No primitive data types (integers, strings, ...) in the λ-calculus;

data is represented by functions (*Church encodings*)

The λ-calculus and Church encodings

A naive syntactic theory of functions:

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

No primitive data types (integers, strings, ...) in the λ-calculus;

data is represented by functions (*Church encodings*)

Idea: $n \in \mathbb{N}$ is encoded as $f \mapsto f \circ \ldots (n \text{ times}) \cdots \circ f$

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

The λ-calculus and Church encodings

A naive syntactic theory of functions:

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

No primitive data types (integers, strings, ...) in the λ-calculus;
data is represented by functions (*Church encodings*)

Idea: $n \in \mathbb{N}$ is encoded as $f \mapsto f \circ \ldots (n \text{ times}) \cdots \circ f$

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

The untyped λ-calculus is Turing-complete

The simply typed λ-calculus

We now consider a *type system*: labeling λ-terms with specifications

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

Simple types: built using "$\to$" from a base type o

The simply typed λ-calculus

We now consider a *type system*: labeling λ-terms with specifications

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

Simple types: built using “$\to$” from a base type o

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

The simply typed λ-calculus

We now consider a *type system*: labeling λ-terms with specifications

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

Simple types: built using "$\to$" from a base type o

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

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

We now consider a *type system*: labeling λ-terms with specifications

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

Simple types: built using "→" from a base type o

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

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

More generally, $t : \mathsf{Nat} \iff \exists n \in \mathbb{N} : t =_{\beta\eta} \overline{n}$

Simply typed functions on Church numerals (1)

Simple types make the λ-calculus terminate: not Turing-complete anymore
$\longrightarrow$ so what can we compute?

Simply typed functions on Church numerals (1)

Simple types make the λ-calculus terminate: not Turing-complete anymore

$\longrightarrow$ so what can we compute? $\quad (t : \mathsf{Nat} = (o \to o) \to o \to o \iff \exists n \in \mathbb{N} : t =_{\beta\eta} \overline{n})$

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`*).*

Simply typed functions on Church numerals (1)

Simple types make the λ-calculus terminate: not Turing-complete anymore

$\longrightarrow$ so what can we compute? $\quad (t : \mathsf{Nat} = (o \to o) \to o \to o \iff \exists n \in \mathbb{N} : t =_{\beta\eta} \overline{n})$

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`*).*

A trick to increase expressive power: for any simple type A, for $n \in \mathbb{N}$,

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

(but in general some inhabitants of $\mathsf{Nat}[A]$ don't represent integers)

Open question

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

Simply typed functions on Church numerals (2)

Open question

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

Why is nobody working on this seemingly natural question?

- Apparently, low hopes for a nice answer until now
  - you can express towers of exponentials
  - but not subtraction or equality (Statman 198X)
- Not so important for actual programming language theory
  - analogy: functional analysis for differential equations vs Banach space geometry for its own sake...

Simply typed functions on Church numerals (2)

Open question

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

Why is nobody working on this seemingly natural question?

- Apparently, low hopes for a nice answer until now
  - you can express towers of exponentials
  - but not subtraction or equality (Statman 198X)
- Not so important for actual programming language theory
  - analogy: functional analysis for differential equations vs Banach space geometry for its own sake… which is closer to infinitary combinatorics than analysis

Simply typed functions on Church numerals (2)

Open question

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

Why is nobody working on this seemingly natural question?

- Apparently, low hopes for a nice answer until now
  - you can express towers of exponentials
  - but not subtraction or equality (Statman 198X)
- Not so important for actual programming language theory
  - analogy: functional analysis for differential equations vs Banach space geometry for its own sake... which is closer to infinitary combinatorics than analysis

Slogan: the above question is not PL theory, it's automata theory!

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

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

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)

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

Theorem (Hillebrand & Kanellakis 1996)

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

Automata theory appears in the simply typed λ-calculus

Theorem (Hillebrand & Kanellakis 1996)

The language $L \subseteq \Sigma^*$ *is* regular $\iff$ *there are a simple type* A *and* $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ *such that* $\forall w \in \Sigma^*,\ w \in L \Leftrightarrow t\ \overline{w} =_\beta \texttt{true}$

Corollary

A simply typed λ*-term of type* $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}$ *defined a function* $f : \Gamma^* \to \Sigma^*$ *which is* regularity-preserving*:* $L \subseteq \Sigma^*$ *regular* $\implies f^{-1}(L)$ *regular*

Automata theory appears in the simply typed λ-calculus

Theorem (Hillebrand & Kanellakis 1996)

The language $L \subseteq \Sigma^*$ *is* regular $\iff$ *there are a simple type* A *and* $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ *such that* $\forall w \in \Sigma^*,\ w \in L \Leftrightarrow t\,\overline{w} =_\beta \mathtt{true}$

Corollary

A simply typed λ*-term of type* $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}$ *defined a function* $f : \Gamma^* \to \Sigma^*$ *which is* regularity-preserving*:* $L \subseteq \Sigma^*$ *regular* $\implies f^{-1}(L)$ *regular*

Another good property: these string-to-string functions are *closed under composition*

$\longrightarrow$ we might expect them to correspond to some *transducer* model!

Automata theory appears in the simply typed λ-calculus

Theorem (Hillebrand & Kanellakis 1996)

The language $L \subseteq \Sigma^*$ *is* regular $\iff$ *there are a simple type* A *and* $t : \mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$ *such that* $\forall w \in \Sigma^*,\ w \in L \Leftrightarrow t\,\overline{w} =_\beta \texttt{true}$

Corollary

A simply typed λ*-term of type* $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}$ *defined a function* $f : \Gamma^* \to \Sigma^*$ *which is* regularity-preserving*:* $L \subseteq \Sigma^*$ *regular* $\implies f^{-1}(L)$ *regular*

Another good property: these string-to-string functions are *closed under composition* $\longrightarrow$ we might expect them to correspond to some *transducer* model!

However, these functions can have grow as fast as any tower of exponentials which is rarely the case for transducers (but precedents exist!)

So, we started out with a "strategic retreat"...

Implicit automata in linear logic (j.w.w. Cécilia Pradic)

Problem: the simply typed λ-calculus is "too expressive". Possible solution: use a *linear* type system $\longrightarrow$ restrict duplication, hence limit growth rate

Implicit automata in linear logic (j.w.w. Cécilia Pradic)

Problem: the simply typed λ-calculus is “too expressive”. Possible solution: use a *linear* type system $\longrightarrow$ restrict duplication, hence limit growth rate

- a common recipe for *implicit computational complexity*: the design of (theoretical) programming languages that characterize complexity classes

Implicit automata in linear logic (j.w.w. Cécilia Pradic)

Problem: the simply typed λ-calculus is "too expressive". Possible solution: use a *linear* type system $\longrightarrow$ restrict duplication, hence limit growth rate

- a common recipe for *implicit computational complexity*: the design of (theoretical) programming languages that characterize complexity classes

Automata theory counterpart: various "single use restrictions"

Several machine models for *regular functions* of strings and trees involve such restrictions [Bloem & Engelfriet 2000; Engelfriet & Maneth 1999; Alur & Černý 2010; ...]

Implicit automata in linear logic (j.w.w. Cécilia Pradic)

Problem: the simply typed λ-calculus is "too expressive". Possible solution: use a *linear* type system $\longrightarrow$ restrict duplication, hence limit growth rate

- a common recipe for *implicit computational complexity*: the design of (theoretical) programming languages that characterize complexity classes

Automata theory counterpart: various "single use restrictions"

Several machine models for *regular functions* of strings and trees involve such restrictions [Bloem & Engelfriet 2000; Engelfriet & Maneth 1999; Alur & Černý 2010; ...]

$\longrightarrow$ λ-calculus characterizations of *regular* and *comparison-free polyregular* functions

+ star-free languages / aperiodic reg. fn. via non-commutative types

+ upcoming work on atoms (with Clovis Eberhart)

also relying on a single use restriction [Bojańczyk & Stefański 2020]

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

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

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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

↓									
a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

	↓								
a	c	a	b	$\#$	b	c	$\#$	c	a

$X = ca \qquad Y = \varepsilon$

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

$$\downarrow$$

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

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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

$\downarrow$

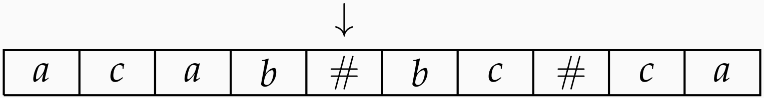
a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

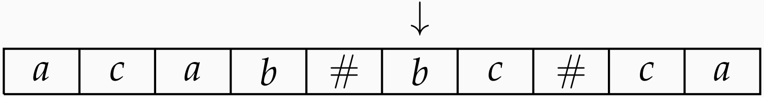


$X = \varepsilon \qquad Y = baca\#$

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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



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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

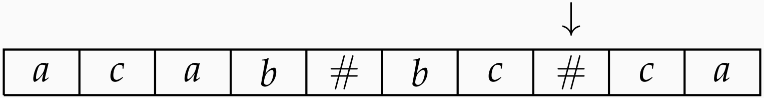
						↓			
a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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



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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

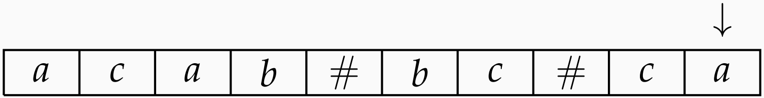
								↓	
a	c	a	b	$\#$	b	c	$\#$	c	a

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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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



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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

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

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

Streaming string transducers [Alur & Černý 2010]

DFA + string-valued *registers*. Example:

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

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

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

Regular functions (a.k.a. MSO transductions) = 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

What happens without linearity?

Copyless streaming string transducers can be encoded in a linear λ-calculus.

What happens without linearity?

Copyless streaming string transducers can be encoded in a linear λ-calculus.

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?

Copyless streaming string transducers can be encoded in a linear λ-calculus.

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 can be encoded in a linear λ-calculus.

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

What happens without linearity?

Copyless streaming string transducers can be encoded in a linear λ-calculus.

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

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

HDT0L transductions

What is known about (compositions of) copyful streaming string transducers?

Theorem (Filiot & Reynier 2017)

- *The much older* HDT0L systems *are isomorphic to "simple" copyful SSTs*
- *Copyful SSTs can be simplified* $\rightarrow$ *they compute* HDT0L transductions

HDT0L transductions

What is known about (compositions of) copyful streaming string transducers?

Theorem (Filiot & Reynier 2017)

- *The much older* HDT0L systems *are isomorphic to "simple" copyful SSTs*
- *Copyful SSTs can be simplified* $\rightarrow$ *they compute* HDT0L transductions

Next, let's search for this keyword in the literature...

HDT0L transductions

What is known about (compositions of) copyful streaming string transducers?

Theorem (Filiot & Reynier 2017)

- *The much older* HDT0L systems *are isomorphic to "simple" copyful SSTs*
- *Copyful SSTs can be simplified* $\rightarrow$ *they compute* HDT0L transductions

Next, let's search for this keyword in the literature...

Theorem (Ferté, Marin & Sénizergues 2014)

The following compute the same string-to-string functions:

- *another notion of HDT0L transduction* = right-to-left (*simple*) *copyful SSTs*
- level-2 pushdown transducers: *see next slide*

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [abc] \end{bmatrix}$$

Output:

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\Longleftrightarrow$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [abc] \\ [abc] \end{bmatrix}$$

Output:

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\Longleftrightarrow$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [bc] \\ [abc] \end{bmatrix}$$

Output:

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ [bc] \\ [bc] \\ [abc] \end{bmatrix}$$

Output:

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ [c] \\ [bc] \\ [abc] \end{bmatrix}$$

Output:

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\Longleftrightarrow$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ [c] \\ [c] \\ [bc] \\ [abc] \end{bmatrix}$$

Output:

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ [] \\ [c] \\ [bc] \\ [abc] \end{bmatrix}$$

Output:

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ [c] \\ [bc] \\ [abc] \end{bmatrix}$$

Output:

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ [] \\ [bc] \\ [abc] \end{bmatrix}$$

Output: c

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [bc] \\ [abc] \end{bmatrix} \qquad \text{Output:} \quad c$$

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [c] \\ [abc] \end{bmatrix} \qquad \text{Output:} \quad cb$$

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [] \\ [abc] \end{bmatrix}$$

Output: cbc

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [abc] \end{bmatrix}$$

Output: cbc

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [bc] \end{bmatrix}$$

Output: $cbca$

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [c] \end{bmatrix}$$

Output: $cbcab$

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

$$\begin{bmatrix} \\ \\ \\ [\,] \end{bmatrix}$$

Output: $cbcabc$

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

Output: $cbcabc$

Pushdowns of pushdowns

Theorem (Ferté, Marin & Sénizergues 2014)

Right-to-left (simple) copyful SSTs $\iff$ *level-2 pushdown transducers*

Let's compute $abc \mapsto (c)(bc)(abc)$

Output: $cbcabc$

Remark: we never need to push sth on the small stacks, they're input suffixes

$\longrightarrow$ "one-way marble" transducers (à la [Douéneau-Tabot, Filiot & Gastin 2020])

Iterated pushdown transducers: using pushdowns of … of pushdowns

We just saw the $k = 1$ case of:

Claim (Sénizergues 2007 — no available proof?)

Composition *of k right-to-left copyful SSTs* $\iff$ *level-$(k+1)$ pushdown transducers*

Iterated pushdown transducers: using pushdowns of ... of pushdowns

We just saw the $k = 1$ case of:

Claim (Sénizergues 2007 — no available proof?)

Composition *of k right-to-left copyful SSTs* $\iff$ *level-*$(k+1)$ *pushdown transducers*

Macro tree transducers [Engelfriet & Vogler 1985] can be seen as bottom-up automata with registers, generalizing right-to-left copyful SSTs to trees.

Theorem (Engelfriet & Vogler 1986 (note the different date))

Composition of k macro tree transducers $\iff$ *level-k* (not $k+1$) *pushdown transducers manipulating pointers to the input tree*

(*provide input as pointer to root, not as stack of letters; pointers can only move downwards*)

Iterated pushdown transducers: using pushdowns of ... of pushdowns

We just saw the $k = 1$ case of:

Claim (Sénizergues 2007 — no available proof?)

Composition *of k right-to-left copyful SSTs* $\iff$ *level-*$(k+1)$ *pushdown transducers*

Macro tree transducers [Engelfriet & Vogler 1985] can be seen as bottom-up automata with registers, generalizing right-to-left copyful SSTs to trees.

Theorem (Engelfriet & Vogler 1986 (note the different date))

Composition of k macro tree transducers $\iff$ *level-k* (not $k+1$) *pushdown transducers manipulating pointers to the input tree*

(provide input as pointer to root, not as stack of letters; pointers can only move downwards)

Note that this directly generalizes the "one-way marbles" ($k = 1$ on strings)

“Engelfriet’s class” of transductions

In fact, the following are equivalent: [Engelfriet & Vogler ’88; Engelfriet & Maneth ’03]

- Iterated pushdown tree transducers (with pointers)
- Compositions of macro tree transducers
  of attribute grammars a.k.a. tree-walking transducers
  of anything in-between (pebble transducers, MSOT w/ sharing, ...)
- “High level tree transducers”: can be viewed as storing *functions* in registers
  (with subtle restrictions, we’ll come back to that)

A quite robust class of hyperexponential transductions...

“Engelfriet’s class” of transductions

In fact, the following are equivalent: [Engelfriet & Vogler ’88; Engelfriet & Maneth ’03]

- Iterated pushdown tree transducers (with pointers)
- Compositions of macro tree transducers
  of attribute grammars a.k.a. tree-walking transducers
  of anything in-between (pebble transducers, MSOT w/ sharing, ...)
- “High level tree transducers”: can be viewed as storing *functions* in registers
  (with subtle restrictions, we’ll come back to that)

A quite robust class of hyperexponential transductions...

Trivial observation

They are included in the simply typed λ-definable functions.

“Engelfriet’s class” of transductions

In fact, the following are equivalent: [Engelfriet & Vogler ’88; Engelfriet & Maneth ’03]

- Iterated pushdown tree transducers (with pointers)
- Compositions of macro tree transducers
  of attribute grammars a.k.a. tree-walking transducers
  of anything in-between (pebble transducers, MSOT w/ sharing, ...)
- “High level tree transducers”: can be viewed as storing *functions* in registers
  (with subtle restrictions, we’ll come back to that)

A quite robust class of hyperexponential transductions...

Trivial observation

They are included in the simply typed λ-definable functions.

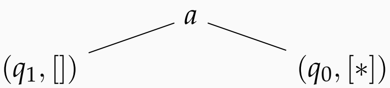
But we’ll see why the converse might fail, via a detour through *infinite* structures

Generating infinite trees

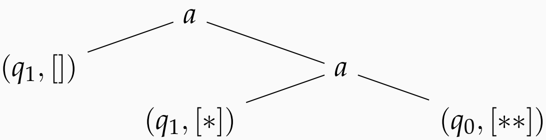
Higher-order pushdown automata = iterated pushdown transducers without input

$$(q_0, [])$$

Higher-order pushdown automata = iterated pushdown transducers without input

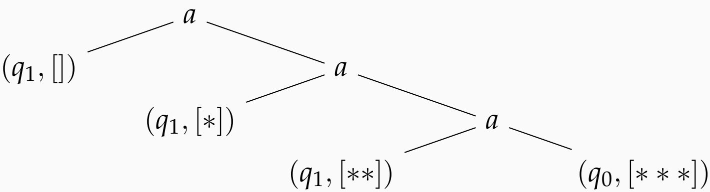


Higher-order pushdown automata = iterated pushdown transducers without input



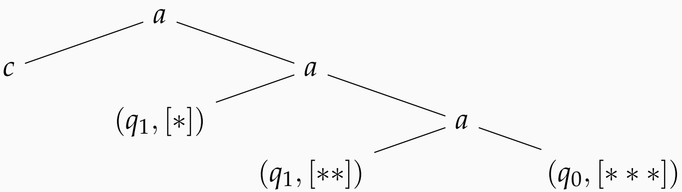
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



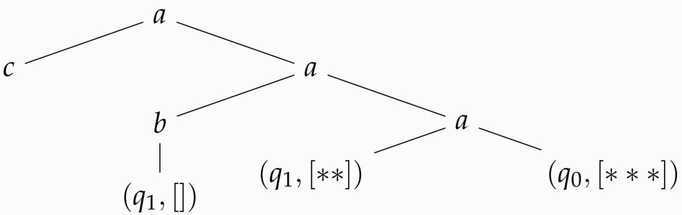
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



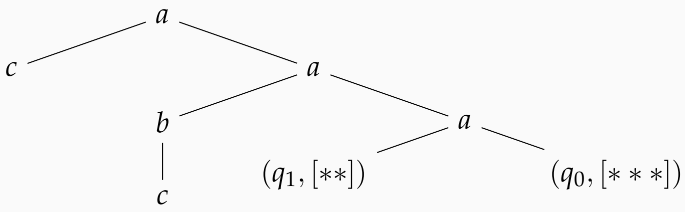
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



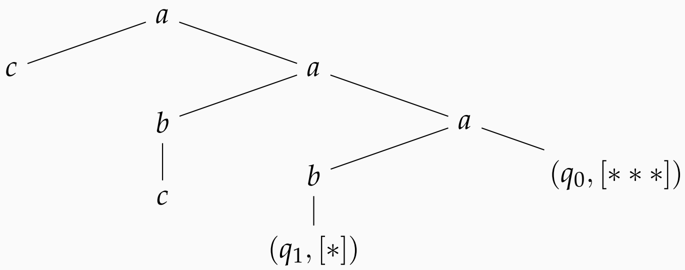
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



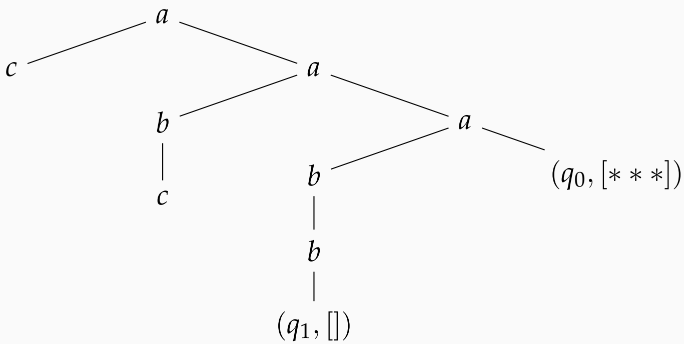
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



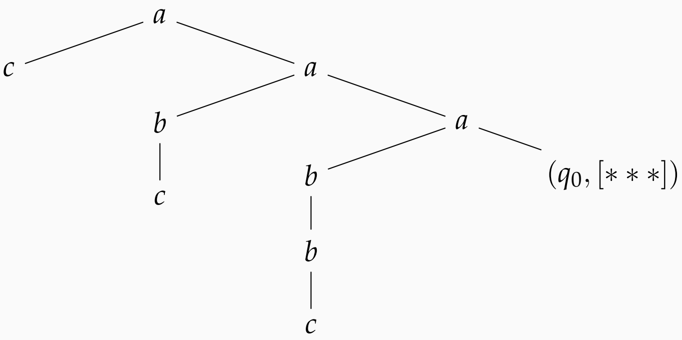
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



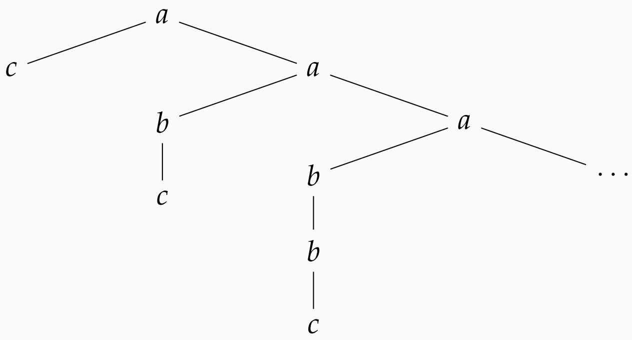
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



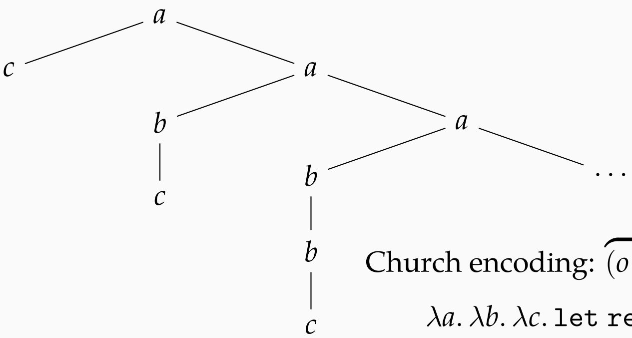
Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input

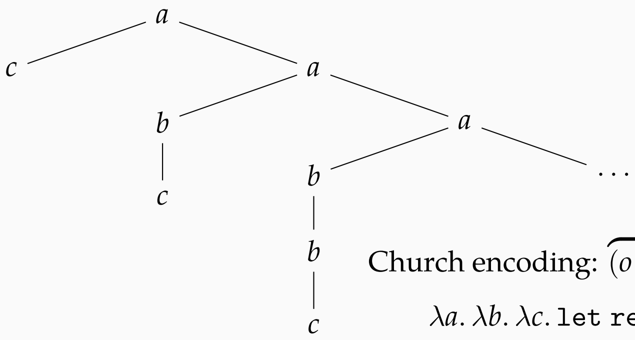


Church encoding: $\overbrace{(o \to o \to o)}^{a} \to \overbrace{(o \to o)}^{b} \to \overset{c}{\overset{\downarrow}{o}} \to o$

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

Generating infinite trees

Higher-order pushdown automata = iterated pushdown transducers without input



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

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

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`

- Safety was first introduced in another equivalent formalism, *recursion schemes*
- Engelfriet & Vogler's "high level tree transducers" are directly inspired from Damm's work on safe recursion schemes

Safely λ-definable functions

Equivalence for formalisms generating infinite trees

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

- Safety was first introduced in another equivalent formalism, *recursion schemes*
- Engelfriet & Vogler's "high level tree transducers" are directly inspired from Damm's work on safe recursion schemes

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

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

Safely λ-definable functions

Equivalence for formalisms generating infinite trees

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

- Safety was first introduced in another equivalent formalism, *recursion schemes*
- Engelfriet & Vogler's "high level tree transducers" are directly inspired from Damm's work on safe recursion schemes

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

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

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

Taking divergence into account

Decomposing the "obvious" theorem

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`
   $\leadsto$ straightforward variant of existing proof [Salvati & Walukiewicz 2012]

Taking divergence into account

Decomposing the “obvious” theorem

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!

Open question

Is there some “manifestly total” machine model for these functions?

More questions on simply typed λ-definable functions

- Can they be obtained by composing significantly simpler functions?

  (recall that this works for the safe case i.e. iterated pushdown transducers)
- Does safety harm expressiveness over trees? over strings? over $\{a\}^* \cong \mathbb{N}$?
- Origin semantics using sets of ... of sets of input nodes?

More questions on simply typed λ-definable functions

- Can they be obtained by composing significantly simpler functions?
  (recall that this works for the safe case i.e. iterated pushdown transducers)
- Does safety harm expressiveness over trees? over strings? over $\{a\}^* \cong \mathbb{N}$?
- Origin semantics using sets of … of sets of input nodes?
- Characterizations of subclasses by growth rate?

Theorem (Engelfriet, Inaba & Maneth 2021)

f computed by an iterated pushdown tree transducer $\wedge\ |f(t)| = O(|t|) \iff f$ *is* regular

Conjecture (Maximality of polyregular functions over strings)

f is simply typed λ-definable $\wedge\ |f(w)| = |w|^{O(1)} \iff f$ *is* polyregular

(*i.e. a composition of polynomial growth HDT0L transductions, see* [Bojańczyk 2018])

Conclusion

We started out by studying the functions definable in the simply typed λ-calculus
(on Church-encoded integers/strings/trees, with input type substitution)

- They (strictly?) include most (all?) known transduction classes, while still falling under the scope of automata theory (definable languages are regular)
- We gave a machine model & raised many questions
- Several connections with recursion schemes & 1980s transducer theory

Conclusion

We started out by studying the functions definable in the simply typed λ-calculus

(on Church-encoded integers/strings/trees, with input type substitution)

- They (strictly?) include most (all?) known transduction classes, while still falling under the scope of automata theory (definable languages are regular)
- We gave a machine model & raised many questions
- Several connections with recursion schemes & 1980s transducer theory

Not the first time typed λ-calculi have led us to a new transducer model!

- Most notably, discovery of comparison-free polyregular (or "polyblind") functions, further studied by Douéneau-Tabot [N., Noûs & Pradic 2021]
- Also: two-way transducers with planar behaviors for FO-transductions

Conclusion

Thanks for your attention!

We started out by studying the functions definable in the simply typed λ-calculus

(on Church-encoded integers/strings/trees, with input type substitution)

- They (strictly?) include most (all?) known transduction classes, while still falling under the scope of automata theory (definable languages are regular)
- We gave a machine model & raised many questions
- Several connections with recursion schemes & 1980s transducer theory

Not the first time typed λ-calculi have led us to a new transducer model!

- Most notably, discovery of comparison-free polyregular (or "polyblind") functions, further studied by Douéneau-Tabot [N., Noûs & Pradic 2021]
- Also: two-way transducers with planar behaviors for FO-transductions